The Blind Software Engineer: Improving the Non-Functional Properties of Software by Means of Genetic Improvement

Submitted for the degree of
Doctor of Philosophy

Bobby R. Bruce

July 12, 2018

Declarations

I, Bobby R. Bruce, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis. Parts of this document are based on the following publications:

- Section 3.1 (in Chapter 3) is based on: Bobby R. Bruce, Justyna Petke, and Mark Harman. Reducing Energy Consumption using Genetic Improvement. In *Proceedings of the 2015 Genetic and Evolutionary Computation Conference — GECCO '15*, 2015.
- Section 3.3 (in Chapter 3) is based on: Bobby R. Bruce, Justyna Petke, Mark Harman, and Earl Barr. Approximate Oracles and Synergy in Software Energy Search Spaces. *IEEE Transactions on Software Engineering*, 2018 (To appear).
- Section 4.2 (in Chapter 4) is based on: Bobby R. Bruce, Jonathan M. Aitken, and Justyna Petke. Deep Parameter Optimisation for Face Detection Using the Viola-Jones Algorithm in OpenCV. In *Proceedings of the 2016 Symposium on Search-Based Software Engineering — SSBSE '16*, 2016; and Bobby R. Bruce. Deep Parameter Optimisation for Face Detection Using the Viola-Jones Algorithm in OpenCV: A Correction. *Research Note RN/17/07, Dept. of Computer Science, University College London*, 2017.
- Section 4.3 (in Chapter 4) is based on: Mahmoud A. Bokhari, Bobby R. Bruce, Brad Alexander, and Markus Wagner, Deep Parameter Optimisation on Android Smartphones for Energy Minimisation — A Tale of Woe and Proof-of-concept. In *Proceedings of the 2017 Genetic and Evolutionary Computation Conference Companion — GECCO Companion '17*, 2017.
- Chapter 5 is based on: Bobby R. Bruce and Justyna Petke. Towards automatic generation and insertion of OpenACC directives. *Research Note RN/18/04, Dept. of Computer Science, University College London*, 2018.

Bobby R. Bruce | Date

Acknowledgements

I want to take the opportunity to thank my supervisory team: Dr. Justyna Petke, Prof. Mark Harman, and Dr. Earl T. Barr. Without their expertise and guidance none of the work described in this thesis would have been possible.

I would also like to give a special thanks to Prof. Bill Langdon whose insightful comments and guidance have aided me greatly, both in the research I have carried out at UCL, and in my career in general.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Life, even in its most basic of forms, continues to amaze mankind with the complexity of its design. When analysing this complexity it is easy to see why the idea of a grand designer has been such a prevalent idea in human history. If it is assumed intelligence is required to undertake a complex engineering feat, such as developing a modern computer system, then it is logical to assume a creature, even as basic as an earthworm, is the product of an even greater intelligence. Yet, as Darwin observed [55], intelligence is not a requirement for the creation of complex systems. Evolution, a phenomenon without consciousness or intellect can, over time, create systems of grand complexity and order. From this observation a question arises — is it possible to develop techniques inspired by Darwinian evolution to solve engineering problems without engineers?

The first to ask such a question was Alan Turing, a person considered by many to be the father of computer science. In 1948 Turing proposed three approaches he believed could solve complex problems without the need for human intervention [167]. The first was a purely logic-driven search. This arose a decade later in the form of general problem-solving algorithms [130]. Though successful in solving toy problems which could be sufficiently formalised, solving real-world problems was found to be infeasible. The second approach Turing called 'cultural search'. This approach would store libraries of information to then reference and provide solutions to particular problems in accordance to this information. This is similar to what we would now refer to as an expert system. Though the first expert system is hard to date due to differences in definition, the development is normally attributed to Feigenbaum, Bachanan, Lederberg, and Sutherland for their work, originating in the 1960s, on the DENRAL system [45, 66]. Turing's last proposal was an iterative, evolutionary technique which he later expanded on [168] stating:

> 'We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution.'

Though a primitive proposal in comparison to modern techniques, Turing clearly identified the foundation of what we now refer to as Evolutionary Computation (EC). EC borrows principles from biological evolution and adapts them for use in computer systems. Despite EC initially appearing to be an awkward melding between the two perpendicular disciplines of biology and computer science, useful ideas from evolutionary theory can be utilised in engineering processes. Just as

man dreamt of flight from watching birds, EC researchers dream of self-improving systems from observing evolutionary processes.

On an abstract level evolutionary biology is the study of how and why genotypes change overtime in order to optimise their respective phenotypes for a given environment. This process parallels itself with software development, where source code is altered to improve a compiled product for a given set of requirements. Each process is carried out on a representation of data and instructions — in biological evolution, DNA; in software engineering, source code. Both are then used to build something which interacts with the 'real world'. DNA creates life, and source code is compiled to produce a software product. Each is evaluated on their suitability to the environment in which they are deployed, with this evaluation feeding back into an iterative cycle to ultimately create systems of greater suitability in the future.

Despite these similarities, evolutionary inspired techniques in computer science have yet to build complex software systems from scratch. Though they have been successfully utilised to solve complex problems, such as classification [65] and clustering [30, 60], there is a general acceptance that, as in nature, these evolutionary processes take vast amounts of time to create complex structures from simple starting points. Even the best computer systems cannot compete with nature's ability to evaluate many millions of variants in parallel over the course of millennia. It is for this reason research into modifying and optimising already existing software, a process known as *Genetic Improvement*, has blossomed.

Genetic Improvement [142] (commonly referred to as 'GI') modifies existing software using search-based techniques [85] with respect to some objective. These search-based techniques are typically evolutionary and, if not, are based on iterative improvement which we may view as a form of evolution. GI sets out to solve the 'last mile' problems of software development; problems that arise in software engineering close to completion, such as bugs or sub-optimal performance. Debugging, for example, takes up a significant proportion of a software development lifecycle. This is astonishing considering how little software requires changing to fix a bug[1]. This is a problem GI has attempted to solve in the subdomain of automatic bug fixing [127]. In automatic bug fixing, the set of possible modifications that may be made to the code is searched to find one that fixes a bug while preserving the remainder of the software's functionality. The output is an 'evolved' version of the code that functions more closely to what the end-user desires. While automatic bug fixing takes the lions share of GI research [142], there have been considerable contributions from other areas within the field. There is no restriction to what can be optimised, assuming it can be formulated into what is known as a 'fitness function'; a function used to guide the GI algorithm over the set of possible software variants to one that is suitable.

While sounding radical, GI is merely research into the continuation of what the software engineering community has experienced since the birth of the discipline. In engineering, there are always two conflicting ideas at play: what is wanted and how it is to be implemented. Much research in software engineering is driven by the idea that you can increase productivity by moving more developer time onto the former by automating the latter. The first software engineers worked on machine code until higher-level programming languages, such as FORTRAN, were introduced in the 1950s. Engineers rapidly flocked to these higher-level languages, understanding that they allowed for greater productivity. The productivity gains are simple enough to understand: programming languages allow few, relatively simple instructions to be processed into many instructions automatically via compilation. Developers thereby code less and in languages which are (hopefully)

[1]The average fix-patch consists of 2.73 changed lines according to a study of the Eclipse and Mozilla code repositories [159].

easier to understand. Furthermore, programming languages enable portability as a language may remain constant while the compiler changes between different computer architectures. Since their introduction, programming languages have grown in complexity, compilers have become a research area unto themselves, libraries have been introduced so developers needn't waste time rewriting commonly used components, integrated development environments have been produced to aid in the management of source code, and analysis tools are now freely available to help developers find bugs and inefficiencies quicker. It is a myth that progress in computer systems has been driven solely by advances in the hardware; the productivity of developers has also increased. While every computer science course will, at some point, cover sorting algorithms, few software engineers need implement their own. `list.sort()` is sufficient. The offloading of tedious tasks to automated processes has been a continuous feature in software engineering and there is no valid reason as to why it should stop. As anyone who has spent any time coding knows, there is still much tedium left to automate.

A good question for any software engineering researcher is, 'where are we heading?'. It is difficult to predict but given the trends we have seen thus far it does not seem too farfetched to imagine a future where developers are even further removed from thinking about *how* things are implemented, to allow more focus on the *what* is implemented. One might imagine a future system in which a software developer simply declares what a piece of code is to do with the implementation left to an automated process. In such a scenario the developer would not worry themselves with, say, how fast his system runs or how much memory it consumes. The system would attempt to give the developer the most optimal solution given the restrictions of his specification and the limitations of the target hardware.

Genetic improvement is the software engineering technology which seems most likely to lead to such a scenario. In particular, genetic improvement of non-functional software properties. Genetic improvement of non-functional properties means, in essence, that the semantics of the targeted code is not changed, but other properties, such as its execution time or energy consumption, are. We may view the semantics of the source code as a by-proxy functional specification, with the GI process eventually returning the most optimal variant of that code to then be deployed. As an example, Manotas et al. [121] optimised the energy consumption of a computer system by swapping Java `java.util.Collection` interface implementations (e.g. `java.util.ArrayList` with `java.util.LinkedList`). Their intuition was, in many instances, developers simply want *a* collection to store data and do not care about which specific one. As Java has no generic collection class, they must select a specific implementation. As each collection implementation can be more or less optimal in different contexts [141], there is a clear opportunity for optimisation. Manotas et al. showed that by searching the space of collection implementations, energy savings of up to 17% are possible.

It is the genetic improvement of non-functional properties, such as execution time and energy consumption, which we concern ourselves with in this thesis, as we find it to be the area of research which is the most interesting, and the most exciting. It is hoped that those referencing this thesis may share the same vision: that the genetic improvement of non-functional properties has the potential to transform software development, and that the work presented here is a step towards that goal.

The thesis is divided into six chapters (inclusive of this 'Introduction' chapter). In Chapter 2 we explain the background material necessary to understand the content discussed later in the following chapters. From this, in Chapter 3, we highlight our investigations into the novel non-functional property of energy consumption which, in part, includes a study in how energy may be

reduced via the approximation of output. We then expand on this in Chapter 4 by discussing our investigations into the applicability of GI in the domain of approximate computing, which covers a study into optimising the non-functional properties of software running on novel hardware — in this case, Android tablet devices. We then show, in Chapter 5, early research into how GI may be used to specialise software for specific hardware targets; in particular, how GI may automatically modify sequential code to run on GPUs. Finally, in Chapter 6 we discuss what relevant work is currently being undertaken by using the area of genetic improvement, and provide the reader with clear and concise take-away messages from this thesis.

Chapter 2

Background

In this chapter we shall outline the background knowledge necessary to understand the improvement of software's non-functional properties using genetic improvement.

2.1 Meta-heuristic Search

In order to understand how software may be automatically modified, one must first understand meta-heuristics. Though the term is often used without regard of a concrete definition, for most intents-and-purposes a meta-heuristic is a general purpose algorithm used to obtain an optimal, or near optimal, solution from a set of possible solutions for a particular problem. It differs from a regular heuristic in that it is not tailored for a specific problem and instead works at a higher, abstract level. Meta-heuristics treat problems as if they are black-boxes which simply take an input and provide feedback on whether the input is good or bad in comparison to other inputs. This feedback is commonly referred to as the 'fitness' of a solution, with the function used to determine this fitness referred to as a 'fitness function'.

As an example, the travelling salesman problem (TSP) is a well-studied optimisation problem in computer science [17]. The problem states that given a graph, a route must be found in which all nodes are visited while minimising the cost of doing so (in the TSP problem, the links between nodes have an associated penalty for traversal, often conceptualised as 'distance'). A heuristic for this problem may start at a chosen node and go to the nearest non-visited node, continuing in this manner until all nodes are visited. As is obvious, this algorithm is specialised to this problem. It cannot be generalised to, say, a knapsack problem [151]. A meta-heuristic is more general. It would view the TSP as a black-box which accepts a permutation of all the nodes as an input. This black box then returns a 'fitness' — in this case the cost of that route through the graph. The meta-heuristic would attempt to find the best permutation from the space of all possible permutations. The meta-heuristic search does not require any detailed knowledge of the problem to be solved. This is good for both portability between different problems and for the solving of problems in which information and generalisations of are difficult to determine.

For a given problem, meta-heuristics search a set of possible solutions. Some solutions are inherently better than others and for any given solution there are likely to be others in the set that are similar and some that are different. Given these two concepts it is common to refer to meta-heuristics as traversing a visual and mental aid known as a *Search Space*. A search space

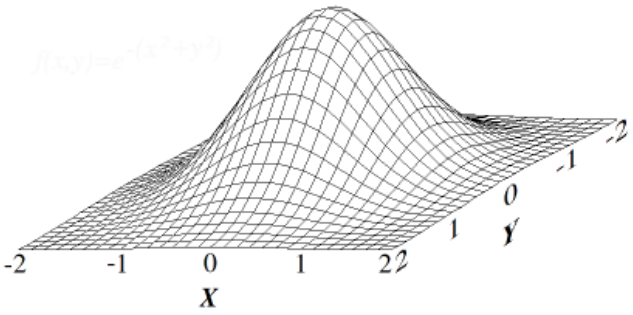


Figure 2.1: A simple search space.

is a theoretical landscape of all possible solutions, inspired by the idea of a 'Fitness Landscape' developed by Wright in 1932 [183]. Typically, similar solutions are nearer each other with the altitude of a solution corresponding to its fitness. Figure 2.1 shows a simple example consisting of two input parameters, X and Y, along with a corresponding fitness value (the unlabelled Z axis, assumed to be unknown prior to search) for each X–Y input pair. It is the role of the meta-heuristic to navigate this search space to find a suitable solution — that is, one that is optimal or near optimal. The question is, how does one go about doing this?

A crude and basic approach to this would be to 'evaluate' all possible solutions and select the best. In meta-heuristic research, an evaluation means giving a search space solution a fitness value, and, in this brute-force approach, means assigning every solution in the search space a fitness value. This would guarantee the best solution is eventually found and, in cases where the number of solutions in the search space is low, it is a sensible option. The downside is, this is often infeasible. The number of possible solutions to many problems is simply too large for all to be evaluated. The purpose of a meta-heuristic is to avoid this brute-force approach and instead offer a compromise where a decent solution is found with a tolerable number of evaluations. The solution produced may not be optimal (in fact, in most cases, it is unlikely to be) but a meta-heuristic allows the process of navigating a search space to be carried out in a reasonable timeframe.

An alternative to a brute-force approach is to uniformly sample the search space. This is known as Pure Random Search [61, 99], a form of Monte Carlo optimisation. The algorithm is simple. First, a random search space coordinate is selected and then evaluated. If the coordinate's fitness is better than the current best (or it is the first to be evaluated) then this coordinate is set as the current best. This process is repeated for as many iterations as is necessary to find a decent solution or as time would permit. This approach has good *exploration* as, if run for a non-trivial number of generations, most areas of the search space will be explored in some capacity. It has however, poor *exploitation*.

In most problems, search spaces are rarely random. Poor solutions are clustered with similarly poor solutions, while good solutions are clustered with other good solutions. This is obvious when you consider that solutions to problems with similar attributes are likely to have similar results. Consider finding a point within the search space that exists somewhere between an optimal peak and a sub-optimal trough. This point may be of high fitness but there exists areas close to it which are of even higher fitness (i.e. up the slope towards the optimal). It is exploitation, a common component in almost all meta-heuristic algorithms, that attempts to find more optimal solutions in

such a scenario. With random search this exploitation can never occur.

If exploitation is required then a hill-climbing algorithm (a basic form of neighbourhood search [97]) may be used. A hill-climbing algorithm starts with a solution within the search space and samples its neighbouring solutions (sometimes exhaustively, sometimes just a randomly chosen subset — this is dependent on the definition of neighbourhood and the setup of the algorithm). The algorithm then 'moves' to the solution of the highest fitness in this sampled neighbourhood. This process repeats until the algorithm reaches a point in which all surrounding solutions are of a lower fitness.

If the hill-climbing algorithm is used alone then it will likely get trapped in a local optima; a location within the search space that appears optimal given the immediate surroundings but is actually sub-optimal when the wider search space is considered. Therefore, a good meta-heuristic requires the use of both exploration, like that found in random search, and exploitation, like that found in hill-climbing algorithms. The most frequently used search algorithms typically strike a balance between the two, allowing the search space to be explored and good solutions exploited.

2.1.1 Simulated Annealing

One such meta-heuristic is Simulated Annealing [169], a technique inspired by the annealing process in which metal is heated above its recrystallisation point then slowly cooled, forming larger crystals and thereby reducing defects. Though the analogy is somewhat stretched, simulated annealing borrows the idea of a 'temperature' that slowly decreases over time. The pseudocode for simulated annealing is shown in Algorithm 1.

Algorithm 1 Simulated Annealing Algorithm Pseudocode.

Require: S_0, a starting location within the search space
K_m, the maximum number of iterations
$\mathit{fitness}(S)$, returns the fitness of a solution, S
$\mathit{neighbour}(S)$, returns a solution, S', that neighbours solution S
$\mathit{temperature}(K, K_m)$, returns a temp t, given the iteration count, $t \in \mathbb{R} \wedge 0 \leq t \leq 1$
$\mathit{random}()$, returns a random number r, $r \in \mathbb{R} \wedge 0 \leq r \leq 1$

1: $S \leftarrow S_0$
2: $f_S \leftarrow \mathit{fitness}(S)$
3: $K \leftarrow 0$
4: **while** $K \leq K_m$ **do**
5: $\quad S' \leftarrow \mathit{neighbour}(S)$
6: $\quad f'_S \leftarrow \mathit{fitness}(S')$
7: $\quad t \leftarrow \mathit{temperature}(K, K_m)$
8: $\quad$ **if** $f'_S > f_S \vee t \geq \mathit{random}()$ **then**
9: $\quad\quad S \leftarrow S'$
10: $\quad\quad f_S \leftarrow f'_S$
11: $\quad$ **end if**
12: $\quad K \leftarrow K + 1$
13: **end while**
14: **return** S

The algorithm starts on a point in the search space S_0. This is normally chosen at random but may be chosen manually if there is a desire to start the search at a particular location. The fitness,

f_S at this location is evaluated via function *fitness*. Then, for a pre-defined number of iterations, K_m, a solution S' neighbouring S is chosen at random (via function *neighbour*) and its fitness f'_S evaluated. If $f'_S > f_S$ then that algorithm 'moves' to the better solution ($S \leftarrow S'$). However, if $f'_S \leq f_S$, moving to the worse solution may be permitted if temperature t is high enough. t is set via function *temperature* which returns high temperature values for early iterations and tends towards zero for later iterations. The temperature, t, can be viewed as the probability of move to a worse solution. Once the maximum number of iterations has been met (i.e. $K = K_m$) then the current search space location is returned.

When the temperature is high simulated annealing exhibits a high degree of exploration and poor exploitation as if it were a random search. Then, as the temperature lowers, the meta-heuristic tends towards having low exploration and high exploitation. When the temperature is zero the algorithm acts as if it were a hill-climbing algorithm. This clever trade-off has been shown effective and has resulted in simulated annealing finding its way into a variety of real-world applications [103].

2.1.2 Tabu Search

Fred Glover was the first to coin the term meta-heuristic [74], and shortly after doing so formalised his own meta-heuristic search technique — Tabu search [75, 76], an enhanced local search technique. While local search (hill-climbing being a typical example) is normally purely exploitative, Glover's Tabu Search approach uses memory structures to allow navigation through worse solutions to escape local optima. The pseudocode is shown in Algorithm 2

The algorithm functions by moving from the current position S to the best position in the set of neighbouring solutions, $\{neighbours(S)\}$. Once a location has been visited it is added to the 'tabu' list L. The word tabu originates from the Tongan language and means something that cannot be touched (where we get the word 'taboo'). The tabu list is a list of locations that cannot be re-visited. The algorithm takes a record of the best location visited thus far, S_B, and returns it when the stopping criteria is met (i.e. the conditions in which the search may cease — typically after a certain number of evaluations or once a solution of adequate fitness has been found). The tabu list is fixed to a maximum length M. When the list is full the oldest member of the list is removed, thereby allowing it to be visited again.

Tabu search acts as a hill-climbing algorithm until hill-climbing ceases to progress then begins to accept worse solutions through a process of exploration.

2.1.3 Evolutionary Computation

As stated in this thesis's introduction, Turing was the first to propose an evolutionary approach to solve complex problems [167, 168]. Though he should be credited as the first to document the idea he did not expand on it, or attempt any investigation into what manner of evolutionary techniques can and should be used to solve complex problems. Therefore, the true pioneer of Evolutionary Computation (EC) is normally cited as being John Holland with his seminal work 'Adaption in natural and artificial systems', published in 1975 [93].

Holland understood that biological evolution is nature's method of traversing a large search space of possible solutions. After all, the variety of life which is theoretically possible is near infinite and, yet, the process of evolution has selected only a small subset to populate the natural world we observe around us. Evolution is neither completely exploitative nor explorative but strikes a balance between the two — an essential property for any decent meta-heuristic. It is for this reason

Algorithm 2 Tabu Search Algorithm Pseudocode.

Require: S_0, a starting location within the search space
M, the maximum length of the Tabu List
continue(), a function which returns false when the stopping criteria is met
neighbours(S), returns the set of solutions neighbouring solution S
fitness(S), returns the fitness of solution S
removeFirst(L), removes the first item from list L

1: $S \leftarrow S_0$
2: $S_B \leftarrow S_0$
3: $L \leftarrow \langle\rangle$
4: **while** *continue*() **do**
5: $S' \leftarrow$`null`
6: **for** $n \in$*neighbour*(S) **do**
7: **if** $n \notin L \wedge (S =$`null`$\vee$*fitness*(n) $>$*fitness*(S')) **then**
8: $S' \leftarrow n$
9: **end if**
10: **end for**
11: $S \leftarrow S'$
12: **if** *fitness*(S) $>$*fitness*(S_B) **then**
13: $S_B \leftarrow S$
14: **end if**
15: $L \leftarrow L^\frown\langle S\rangle$
16: **if** $|L| > M$ **then**
17: *removeFirst*(L)
18: **end if**
19: **end while**
20: **return** S_B

evolutionary processes have provided so much inspiration in the development of meta-heuristic algorithms. These algorithms form a field known as 'Evolutionary Computation' (EC).

Figure 2.2 shows a basic model of how evolution is typically carried out within EC. The process is simple — a subset of solutions to a problem is first translated into a representation that can be mutated and participate in crossover (we will go into detail about crossover and mutation later in this section). This representation varies from problem to problem. It may be a vector of values as originally outlined by Holland [93] or an abstract syntax tree as is common in genetic programming [25]. Regardless, a population of solutions are translated to their respective representations to serve as the initial population. Each representation is evaluated to determine their fitness, thus allowing the individuals in the population to be sorted from best to worst.

Once this initial population is evaluated the best members are selected and the others discarded. These surviving individuals participate in crossover, producing 'offspring' in a similar manner to breeding in nature. Mutations are then added to the population. Mutations may be added to just the children or to all individuals; such decisions are determined by those implementing the algorithm. Once complete, the population is then re-evaluated and the process continues until a stopping criteria is met — often once this loop has iterated for a predetermined number of times. These 'loops' are commonly referred to as 'generations'.

Evolutionary techniques are difficult to discuss in a general sense. Approaches to evolutionary computation are generally tailored for the problem they seek to solve and, as a result, there exists no global EC algorithm that is the best suited for all problems. For example, steady state genetic algorithms [161] completely discard the notion of a generation and, instead, weaker members of the population are gradually replaced with offspring and mutants of fitter members. Regardless, here we discuss two attributes of EC that are common — mutation and crossover.

Mutation

Within biological systems it is estimated around 70% of mutations are harmful [152], with the vast majority of the remainder likely to have no effect. Though the same statistics cannot be applied to the EC community due to vast differences in representation, it is established that mutation is not without risk; it can be destructive, often doing more harm than good to a solution. Despite this risk, the process remains central to almost all systems that utilise evolutionary techniques.

In both EC and biological evolution it is understood that without mutation populations tend towards stagnation. This observation is key to understanding Darwinian evolution [55]. Evolution without mutation is simply a case of recombining genetic material already present within the population. When the optimal combination of this limited genetic material is found, there can be no more improvement.

There have been debates on the role mutation plays. Holland argued mutation's role was to re-introduce genetic material that may have been lost via crossover [93]. He also argued that crossover was the most important element when utilising evolutionary techniques. This is likely due to Holland's belief in the *Building Block Hypothesis* [93, 77] which states that complex solutions are built up using discrete functional units. Therefore, crossover is the most important operator as it allows fit solutions to exchange these blocks to produce potentially more optimal solutions. Though there is little evidence to dispute the hypothesis it is none-the-less a paradigm; a way in which we can view and understand the evolutionary process but not one that is all-encompassing. Hinterding et al. [91] argued that the mutation operator is of greater value than Holland suggests. He observed many cases where the mutation operator was superior to crossover in obtaining more

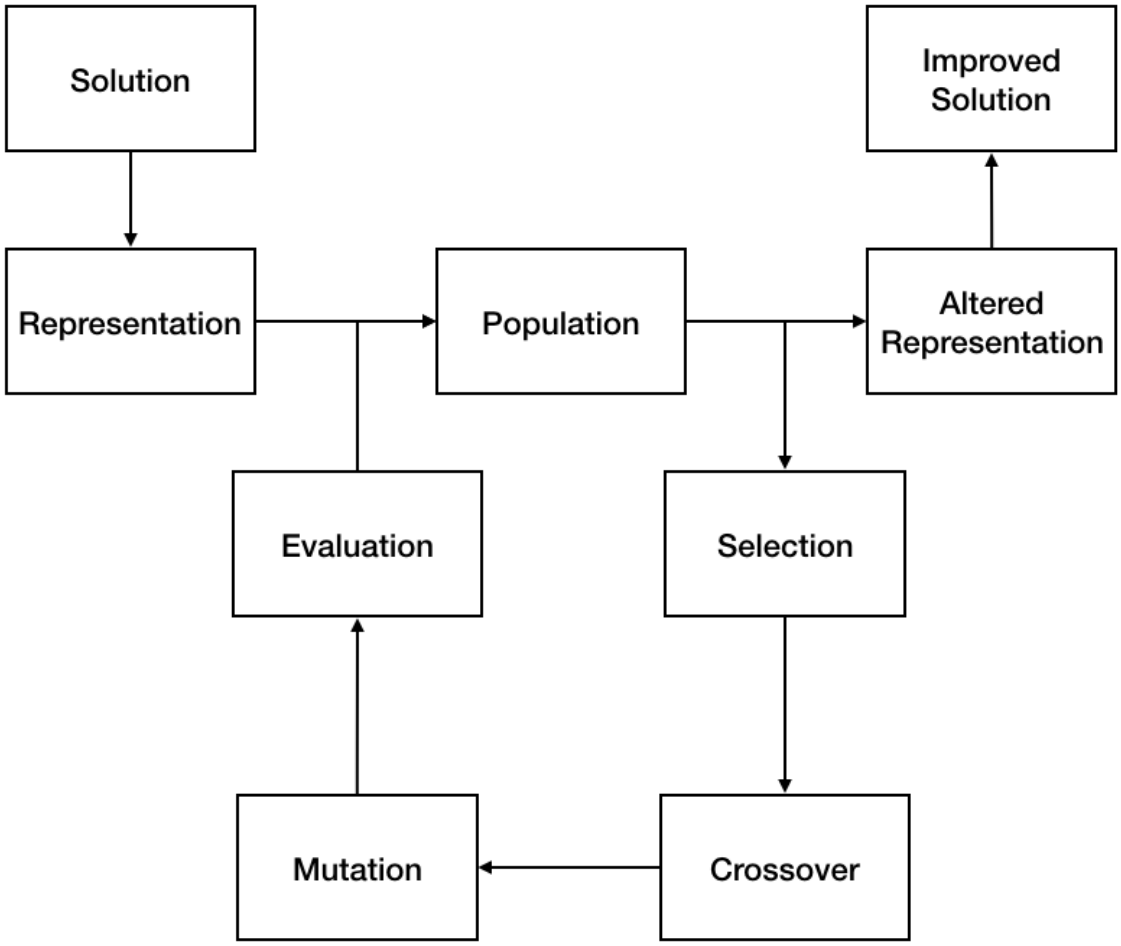


Figure 2.2: An outline of Evolutionary Computation.

optimal solutions to a problem.

There are no rules on what a mutation operator can be, it is entirely dependent on how the problem to be solved is represented and is, ultimately, decided by the EC practitioner. In a binary array representation, a mutation operator would likely randomly choose a position within the array and toggle the bit, while in genetic programming, where tree-based representations are common, a mutation operator may replace a subtree with a randomly generated replacement (this is known as Whigham mutation [176]). Typically a mutation operator makes small changes, permitting the navigation of the local search space in an exploitative manner. However, this is dependent on the mutation operator chosen and the problem representation. For example, in Grammatical Evolution [149], a popular evolutionary approach, a mutation to the representation can produce a radically different solution, leaving some to question whether it is much better than pure random search [177].

Crossover

Crossover, like mutation, is a concept borrowed from biological evolution though the term itself is used more loosely in EC than in the biological sciences. While within biology, crossover is a specific process that occurs during meiosis, in EC it is used to refer to what we can think as breeding; taking two (or in some special cases more than two [165]) solutions to produce a child or children.

Within biology the purpose of breeding has been widely discussed. Mating is costly for individuals while producing clones (as is the case of single-celled organisms, as well as some insects and plants) is cheap. In which case, why is breeding common? In 1889 August Weismann theorised that breeding maintained variance [175] (an idea later expanded upon in genetic terms by Muller [129] and Fisher [67]) as breeding between two individuals produces different children each time thus ensuring a population is sufficiently diverse. As previously discussed, maintaining diversity in EC is important. Without it the potential solutions that can be found becomes limited by lack of unique genetic material.

Another advantage of breeding is that good mutations can travel through a population with greater ease than simply cloning good solutions [32]. If two individuals within a population contain two different advantageous mutations, breeding allows the formation of an individual which contains both. While, in a situation without breeding, each individual would have to rely on the chance that their future generations eventually develop the same advantageous mutation of the other.

While biologists must concern themselves with properties of genetics such as genotypes consisting of paired genomes, complicated further with phenomenons such as dominant and recessive genes, in evolutionary computation this model can be simplified. In genetic algorithms, for example, the most common representation of a genotype is a vector of values which allows for simple crossover. Crossover operators are tightly coupled with the representation chosen.

Figure 2.3 shows a basic one-point crossover working on two binary array representations. For the purposes of this example, we may assume the problem we are solving is 'max-ones' — a toy-problem where the fitness function is the number of elements in the array that are true (i.e. '1'). In Figure 2.3, both parent 1 and parent 2 have a fitness of 4. The crossover takes the first half of parent 1, and the second half of the parent 2, to produce a child. In this example, this child is a solution with a fitness of 8 and is optimal for this problem.

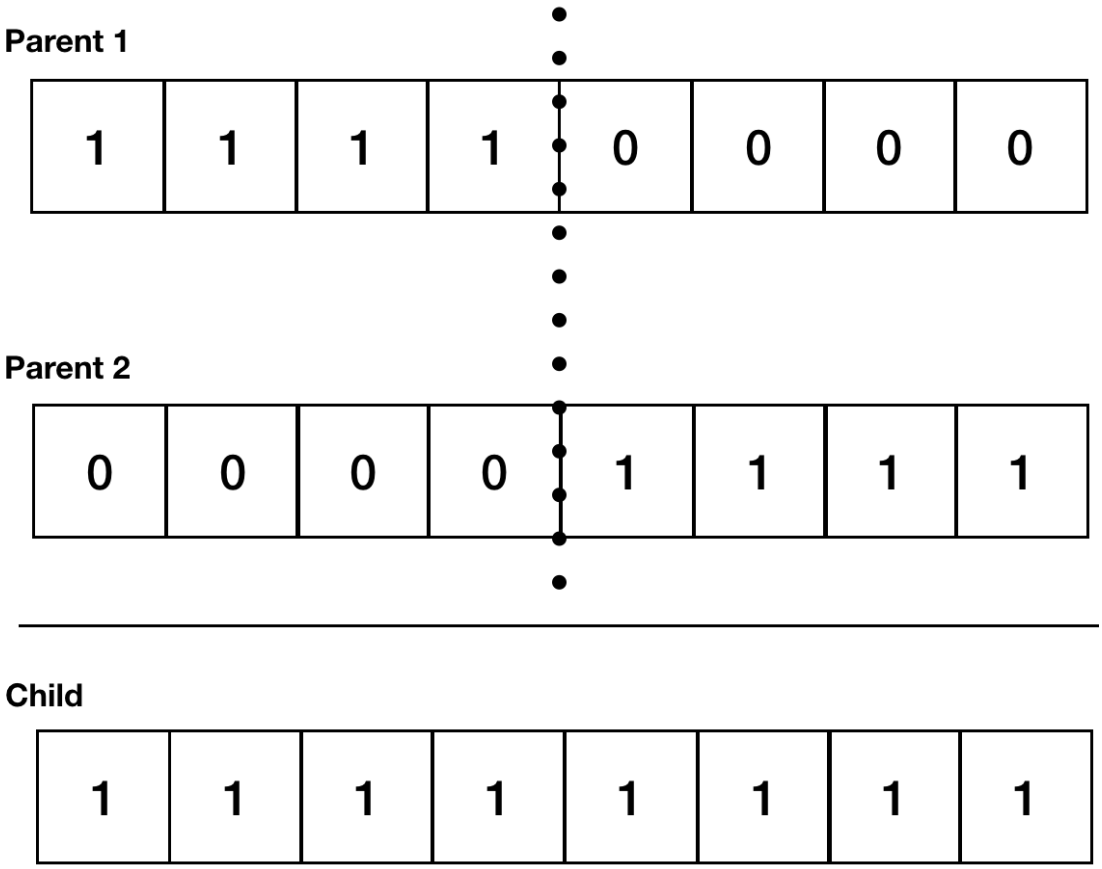


Figure 2.3: A simple one-point crossover between two binary arrays.

2.2 Search-based Software Engineering

In 2001 Harman and Jones coined the term 'Search-based Software Engineering' [85] (SBSE), realising that many problems in software engineering were optimisation problems, and that, at present, these problems are solved by expensive human labour which could be automated via the incorporation of meta-heuristic search techniques. Since SBSE's introduction there has been exponential growth in the field [87]. It has extended to all areas of software engineering with research into concepts such as mutation testing to evaluate the suitability of software tests [96], automated requirements engineering [189], and the automatic tuning of software parameters [48]. Each employ meta-heuristics to navigate a search space of solutions to aid in the development of software. At this point in time goal is not the complete automation of software development but to improve developer productivity through the introduction of new tools and techniques.

Despite its relative youth, some techniques from this field have made their way into 'real world' applications. Fraser and Arcuri developed Evosuite, an automatic test suite generation tool [69]. It uses evolutionary techniques to create a test suite which generalises program behaviour; thereby automating the process of creating regression tests. Evosuite is, at the time of writing, supported by Google and has been used in a variety of open-source and industrial projects [4]. Sapienz, a search-based multi-objective automated testing framework for Android applications [122] is now used by Facebook to find bugs in their mobile applications [16]. In the field of bug fixing, a system has been developed which generates test data based on user interactions, records crashes and attempts to fix them during downtime [83]. Initial trials look promising. Over a 6 month period, in which the system was deployed to monitor a data management application at a rehabilitation centre, the system located and fixed 22 bugs. Despite these impressive impact stories, research is still being taken to discover what the limitations are and what is achievable in the exciting new area of software engineering.

2.2.1 Genetic Improvement

In this thesis we focus on genetic improvement. Genetic improvement, GI, is a SBSE technique which uses meta-heuristic search to modify software with the goal of improving it with respect to a given criterion. A survey by Petke et al. in 2017 (surveying GI papers up to the end of 2015) found there has been a significant growth in this area in recent years from no more than 6 core publications a year prior to 2013[1], to 24 in 2015 (more than double that published in 2014) [142].

The earliest example of genetic improvement[2] was carried out by Ryan and Walsh who, in 1995, used genetic programming to convert sequential programs into parallel equivalents [172] to reduce execution time. The field largely remained dormant until 2008 when Yao and Arcuri carried out an investigation into using evolutionary techniques to fix bugs [18]. The investigation attempted to fix faulty versions of a bubble-sort algorithm using a co-evolutionary approach where tests evolved alongside the target software. The tests evolved to find bugs while the software evolved to pass the tests. This resulted in the majority of bugs being successfully fixed.

Genetic improvement is a vast field of research, covering areas as diverse as the automatic migration of one piece of code into another system (a process known as 'software transplantation' [27]) to reducing software down to its most minimum and necessary components (known as 'software slim-

[1] Petke et al. [142] admit the boundaries of genetic improvement are not well defined and, therefore, their survey may exclude work considered by others to be genetic improvement.

[2] As surveyed by Petke et al. in their 2017 survey [142]

ming' [106, 185]). It can be difficult to divide into meaningful categories, while avoiding so-called 'grey areas'. In this work we divide the field into two major, simple to understand, subdomains: genetic improvement for the improvement of functional properties (where softwares' semantics are changed, as to in automated bug fixing), and genetic improvement for non-functional properties (where softwares' semantics are preferably maintained and properties such as execution time or memory consumption are optimised). In the research outlined in this thesis, we focused on the latter. However, we appreciate that the former contains work that is still of value. Within the following subsections we discuss these two faces of GI, highlighting noteworthy research or research necessary to understand the research presented in this thesis.

Genetic Improvement of functional properties

Improvement of functional properties is probably the most interesting subdomain in GI. It touches upon the dream of the complete automation of software development. In fact, the field has its roots in Automatic Programming; a wide-ranging field with the goal of producing programs from high-level specifications. Automatic Program Synthesis is a subdomain of this field where programs are derived from formal specification of the program's input and output predicates. In their 1971 paper, 'Towards Automatic Program Synthesis', Manna and Waldinger [119] outlined how such a framework may function using automatic theorem provers (by their own admission, their approach lacks technical details). Their approach starts with input and output predicates, provided by a human developer. From this a theorem is induced, and this theorem is automatically proven. A program is then extracted from the proof. An example of this would be the following[3]:
If we wished to automatically synthesise a program that accepts two numbers, X_1 and X_2, and returns the quotient Z_1, and the remainder Z_2 (where $X_2 \neq 0$), we would first formulate our input predicate:

$X_2 \neq 0$

We would then define our output predicate:

$(X_1 = Z_1 \cdot X_2 + Z_2) \wedge (Z_2 < X_2)$

From these a theorem would then be induced:

$(\forall X_1)(\forall X_2)[X_2 \neq 0 \wedge (\exists Z_1)(\exists Z_2)[(X_1 = Z_1 \cdot X_2 + Z_2) \wedge (Z_2 < X_2)]]$

Finally, an automatic theorem prover would then prove the theorem and a program would be exacted from the proof.

There is, of course, much left out of this which would be necessary for implementation: how is the theorem prover to be constructed?; what real-world language could such a system target?; what 'primitives' would be permitted during construction; can this technique be scaled?; and is it an efficient approach given the cost of requiring human-written predicates? Despite these questions, the approach gives an idea of how the functional properties of a program may be synthesised.

While complete automatic synthesis of software is, presently, not a feasible goal of genetic improvement, Manna and Waldinger state at the end of their 1971 paper:

> 'Another way program synthetic techniques may be used in the improvement of an already existing program is in the construction of an automatic debugging system. Current program verification methods give us a way to detect and locate errors in a program; we then can use the program-synthetic approach to replace the incorrect segment without affecting the remainder of the program.'

[3]This is based on Example 1 from 'Towards Automatic Program Synthesis' by Manna and Waldiner [119].

This is interesting as it shows an early attempt to focus on the automatic repair of functional defects. While the particulars have varied, the main elements in modern automated program repair techniques, remain the same. With a sufficiently robust definition of correctness, an automatic approach may be deployed and, guided by this definition, rectify errors. Arcuri and Yao, in 2008, used genetic improvement in such a manner [18], borrowing heavily from both the automatic program synthesis community and evolutionary computation. The goal of their work was to develop a bug fixing framework where a faulty program is provided along with a formal specification of how the program should function (like that required for automatic program synthesis) and the program would be fixed. Their approach used co-evolution where a test suite was generated from the specification, with the fitness of their effectiveness dependent on if they found bugs in the program. The program, on the other hand, evolved to pass as much test cases as possible, by making modifications to the program's abstract syntax tree. Once a program was developed in which no failing test case could be produced, the program was considered fixed.

In their work they fixed a sorting algorithm. Their specification simply stated that for an input vector $V = \langle V_1 \dots V_N \rangle$, the output vector V' must obey the following: $\{V'_1 \dots V'_N\} = \{V_1 \dots V_N\} \wedge (\forall i \in (2 \dots N))[V'_{i-1} \leq V_i]$, (assuming an ascending order output). This approach to bug fixing worked to an extent. Over 100 trials, the approach was capable of fixing 5 for the 8 bugs introduced. Though interestingly, there are more advanced software repair tools available that do not depend on a formal specification being generated — a task known to be costly [105]. In 2010 Wilkerson et al. created a proof-of-concept for a similar approach that used a fitness function instead of a formal definition of program semantics [182].

GenProg, another automatic software repair tool, took a different approach [112]. It used traditional unit tests to locate, then verify the correctness of software modifications. The advantage of this is that a failing test is the traditional means as to how developers find code is incorrect. GenProg therefore uses the developers' own measure of correctness as a guide. The disadvantage of this approach is it requires an adequate test suite to be present; a requirement that is not guaranteed. GenProg can modify the program in three ways: it can *delete* a statement, *insert* a statement above another statement, or *swap* a statement with another. These operators modify the software by using resources contained within it (i.e., *insert* and *swap* operators, can only use statements contained within the target source code). This is an example of the 'Plastic Surgery Hypothesis' which, as outlined by Barr et al. [26], states that:

> 'Changes to a codebase contain snippets that *already exist in the codebase at the time* of change, and these snippets can be *efficiently found* and exploited.'

In short, source code is sufficiently redundant [70, 108] to the extent that material necessary to produce an improved result is likely to be contained within the source code itself. GenProg exploits this hypothesis by finding repairs using statements found within the software being repaired.

These modifications to the source code are applied to statements proportional to their occurrence in test execution traces (i.e., those statements which occur more frequently in failing test execution traces are more likely to be modified). With this, GenProg uses a meta-heuristic search with the goal of minimising the number of failing tests.

GenProg was evaluated on 16 C programs (each of which contained a defect), with the total set of programs consisting of 1.25M LoC. The tool was capable of fixing all bugs in an average of 357 seconds per defect. Later, Le Goues et al. updated GenProg [111], improving its design to scale to larger applications, and exploited the inherent parallelism of its design (i.e. program modifications to an application may be evaluated independently, in parallel). To evaluate this update, they

targeted 8 real-world applications, containing over 5.1M lines of code, and 10,193 test cases. They attempted to fix 105 bugs across these applications, previously fixed by the developers (thereby giving GenProg 'real-world' bugs to fix). They found GenProg was capable of fixing 55 of the 105 bugs found and that, when run on an Amazon EC2 instance of the time (2012), the bugs could be fixed at an average cost of $7.32 (USD) each.

GenProg has been criticised by Qi et al. [148], who stated that, in the original GenProg work, the majority of the generated patch 'fixes' were, in reality, simply deleting functionality in such a manner that tests passed. To demonstrate this, Qi et al. developed Kali. Kali is a repair framework, similar to GenProg, that only deletes functionality. They found Kali 'fixed' at least as many bugs as GenProg (as defined by the test cases), and similar 'generate-and-validate patch generation systems' such as RSRepair [147] and AE [173]. This highlights that having a good program oracle, which can sufficiently describe program behaviour, is essential. In the case of GenProg, this means a better test suite.

There are alternatives to providing a good test suite, or providing a formal specification of how a program may function. Haraldsson et al. developed an approach which utilised real user input data to fix bugs [83]. Their GI framework, during the usage of a targeted system, logs input instances that result in an exception being thrown. Then, when the system isn't in use (at night, for example), a genetic improvement algorithm is used to fix the errors recorded. First, test cases are created by taking the failing inputs and generating close variants. If these variants also resulted in failures (i.e. an exception being thrown) they are added to the test suite. Once an adequate suite of failing tests is generated, a GI algorithm is run on the software, modifying it until all the test cases pass. In their approach, the GI algorithm can modify the code by altering numerical constants, arithmetic operators, arithmetic assignments, relational operators, logical operators and logical constants. After finding a solution which passes all the failing test cases, a report is developed, highlighting the fix that is required, which is delivered to a developer for verification and implementation. This human-in-the-loop approach enables developers to feel more confident with automatically generated software. When deployed to a rehabilitation centre's IT system (written in Python 2.7, consisting of over 25K LoC), Haraldsson et al.'s GI system was able to detect and fix 22 bugs over a 6 month period.

Though much research into optimising software's functional properties focuses on the fixing of bugs, there have been attempts to expand functionality. In 2014, Harman et al. developed a technique known as 'grow and graft' [84] where functionality required is 'grown' in isolation then 'grafted' into the target application. In the growth phrase, the developers give the framework some libraries/APIs that may be useful for the task at hand, as well as a test suite that sufficiently specifies how the component is to behave. The component is then grown in an evolutionary manner. Once the component has fully developed it is grafted to the application. The challenge during the grafting stage is to search the space of insertion points and how the input/output behaviour of the grown component is to interact with the target application. A test suite is provided to define this behaviour. In their work, Harman et al. [84] targeted Pidgin, a C/C++ instant messaging platform consisting of over 200K LoC. They attempted to implement a feature for translating English text to Portuguese and Korean. To do so, during the graft stage they gave the system access to `Google Translate` APIs. Running their framework they were able to create a variant of Pidgin (which they called 'Babel Pidgin') that contained the successfully implemented functionality. We may view Harman et al.'s work on Pidgin as a proof-of-concept. Their 'grow and graft' approach to GI was very specialised to the creation of Babel Pidgin, and clearly requires generalisation before being deployed elsewhere.

This 'grow and graft' approach was inspired by work discussed in a 2015 paper called 'Automated Software Transplantation'[4] [27], where the μScalpel tool was introduced. Instead of 'growing' a solution, μScalpel takes code already functioning in the context of another application. This functionality is then 'transplanted' into another 'host' system. The μScalpel approach attempts to overcome the challenges a human developer would encounter when carrying out such a task such as locating the code to be transplanted in the 'donor' application; finding the correct insertion point in the 'host' application; and altering the variables in the transplanted code so the application functions as intended. In the paper, μScalpel is tested by transplanting code from five 'donors' to three 'host' programs. The authors found four of the five features could be successfully transplanted. In a more in-depth case study, μScalpel was used to transplant a new encoding format (H.264) to the VLC media player. The transplant was successful and took just 26 hour of computation time, passing all regression and acceptance tests. In a later paper [123], μScalpel was used to translate call graph features into KDE's Kate text editor.

Though only a taster of what the domain has to offer has been discussed here, genetic improvement of functional properties is still very much in its infancy. Genetic improvement techniques cannot generate fixes for all bugs, and features cannot be automatically implemented in all cases. However, it seems likely that these approaches will be improved upon, to the benefit of the software engineering community.

This chapter has purposely been written to decouple improvement of the functional, and improvement of the non-functional. Though, in reality, the two areas are inspired by each other. GenProg, for instance, could easily be reconfigured to improve a software system's execution time. All that would be required would be a re-write of the fitness function to minimise execution time while preserving semantics (which could be defined, by-proxy, via a regression test suite). A similar modification could be made to the Haraldsson et al.'s GI system — instead of capturing errors from user input, the system could capture inputs which resulted in the system running in an unacceptably long period of time, and alter the system accordingly. Transplantation could also be modified to replace inefficient components with more efficient ones. So, while different, and not the primary focus of this thesis, optimisation of functional properties via genetic improvement can still inspire techniques to improve non-functional ones.

Genetic Improvement of non-functional properties

If we are to take two products, from any area of engineering, which deliver the same functional usefulness, it is tempting to say they are effectively equivalent. Two bridges, for example, which traverse the same distance and permit the same tonnage of traffic may be said to be equal in value. However, such an analysis fails to take into account what we commonly refer to in software engineering as 'non-functional properties'. The bridges, for instance, may have different maintenance costs. One may be more easily modified to accept more traffic, or one may simply be safer, given a sufficient definition of safety. Properties such as these are usually harder to specify and are often in conflict with one another in ways that are hard for those providing the requirements to fully actualise (features which make a bridge safer may increase maintenance costs, for example). These non-functional properties are difficult to handle. While functional properties require developers to think about *what* is to be implemented, non-functional properties require developers to think about

[4]Though the 2014 'grow and graft' paper was published earlier, the research outlined in the 2015 paper was already complete in 2014 (awaiting publication) and was known about by those carrying out the research into the growing and grafting of functionality.

how something is to be implemented.

In software, if developers wish to optimise for non-functional properties they must keep in mind the multitude of different ways the same piece of functionality may be implemented, what the non-functional properties of these variants are, and how these non-functional properties may trade-off with one another. For example, the C dynamic memory allocation function, '`malloc`', is one in which the majority of C programmers use without thought. They are happy to offload the task of memory allocation to a library they trust. However, Wu et al. observed that the `malloc` function can be optimised to decrease the execution time of target applications [184] (up to 12%). This decrease, however, is not free — it comes at the cost of increased memory consumption. The developer must decide whether this trade-off is acceptable. It may be in cases where memory is plentiful and quick processing is necessary. Of course, there are scenarios where the opposite may be true (Wu et al. also observed that memory consumption could be reduced by up to 21% at the cost of execution time). It is the role of genetic improvement of non-functional properties to automate this process, removing the burden from developers, thus giving them more time to focus on functional concerns.

In an early work in the area, Langdon and Harman attempted to optimise GZip, a popular compression tool [107]. They called their technique 'Genetic Interface Programming' though, as the process involved optimising a piece of code (albeit a small piece), this can be classified as genetic improvement. Their investigation introduced a specialised grammar (loosely based on BNF), allowing certain structural elements to be preserved during the GI process, thus reducing compilation errors. They targeted a piece of sequential C code in GZip and altered it to run efficiently as a CUDA kernel on a GPU. This work is notable as it demonstrated the idea of using the original software as an 'oracle' to verify output. An oracle is a construct that verifies the semantic correctness of a piece of source code [28], and typically (in the field of GI) takes the form of a well-designed test suite. While in domains such as automated bug testing, an oracle must specify how a correct piece of software is to behave (as opposed to the current, presumably incorrect, piece of software), genetic improvement of non-functional properties does not suffer from this problem. As the semantic properties of the software are to be preserved, the original, unmodified, code can be used as reference [174].

In 2011, White et al. presented a framework for 'evolutionary improvement' of software [178]. Their insight was to use genetic programming — an evolutionary-based, meta-heuristic algorithm which synthesises a program (normally in the form of an AST, or similar) in accordance to a fitness function. The emergence of genetic programming is normally attributed to Koza who developed the technique in the early 1990s [104], and has since been used to create programs to solve a wide variety of problems [13]. While genetic programming typically starts from scratch (i.e. an empty tree), White et al.'s insight was to carry out the standard genetic programming procedure on a tree already representative of a human-written program. Creating trees for eight small programs, they were able to reduce the number of executed instructions (their proxy for execution time) by up to 98.6%. Later work in genetic improvement, such as the GenProg bug repair tool [112], adopted this genetic programming-based approach. In White et al.'s paper, the authors admit the technique still required further work. In particular they highlighted three areas of concern: 1) the necessity of ensuring the semantic correctness of the solutions produced (the oracle problem); 2) the problems of scaling to larger applications; and 3) whether different, or multiple non-functional properties may be targeted. These problems are still much discussed today.

To tackle the criticism that genetic improvement was only suitable for small programs, Langdon and Harman attempted to optimise Bowtie2, a widely-used DNA sequencing system, consisting of

around 50,000 lines of code (in which 20,000 were targeted for improvement) [108]. Similar to their 2010 work on Gzip, they used a grammar to represent the lines of code, and thus enforced rules on how (and which) lines could be modified. In practise this meant that statements, `IF/WHILE/FOR` conditions, and `FOR` pre- and post-conditions could either be *deleted*, *copied*, or *replaced* with each other, in accordance to the Plastic Surgery Hypothesis. To overcome scalability issues, Langdon and Harman carried out 'sensitivity analysis' on the software. This sensitivity analysis was similar to traditional software profiling techniques, such as `gprof` [79], in that, based on test inputs, software components (in Langdon and Harman's case, lines of code), were ranked by execution frequency. With this information the search was biased towards those which were executed more often (they were chosen for modification with a higher probability than those of a lower weight). Optimising Bowtie2 with this approach, they were able to make the program 70 times faster — an impressive result by any measure. Additionally, they were also able to improve the accuracy of the algorithm in some cases. Bowtie2 inherently trades-off accuracy for a reduction in execution time as accurate DNA sequencing is too costly for large datasets. As Langdon and Harman tested against a dataset where the accurate 'ground truth' results were known, they ensured the improved code could not produce worse solutions, on average, than the original source code though improvements in accuracy were permitted. In this case, the tool found an optimisation that improved the accuracy while reducing execution time. Though Langdon and Harman permitted a positive change in functional correctness while reducing execution time, others have permitted functional correctness to degrade if an acceptable optimisation of non-functional properties was observed. This is part of an emerging area of computer science called *Approximate Computing* [81], where outputs may be approximated if doing so benefits other properties.

In 2011 Sidiroglous-Douskos et al. [156] used a very simple algorithm to create approximate variants of software targets with reduced execution time. In their approach, Sidiroglous-Douskos et al. modified the amount of loop iterations within a program. A `FOR` loop, for example, `for(i = 0; i < b; i++)`, was modified so its post-condition could be scaled. I.e., in our example, `for(i = 0; i < b; i+=n)` where `n` is tuned to reduce execution time. In Sidiroglous-Douskos et al.'s work it was found that an application's execution time could be decreased by a factor of seven while degrading output quality by less than 10%.

Sitthi-Amorn et al. [158] attempted to use approximate computing to reduce the execution time of shaders by permitting faults in the graphics they produced. They represented the shader code as an abstract syntax tree which they then modified in a genetic programming inspired approach. Figure 2.4 shows an example from their work on different outputs from a modified shader. As can be seen, the original (which runs in 3.16 ms) can be modified to produce variants with a slight error but with a significantly reduced execution time. In this example, Sitthi-Amorn et al. found a variant that ran in 0.73ms (a 76.69% decrease), with an error that is near imperceivable.

In 2014 Li et al. reduced the energy consumption of OLED Smartphones by decreasing the quality of an Android application's user interface [116]. The energy consumption of an OLED screen is correlated to the RGB value of each pixel. Therefore, the goal in optimisation of energy consumption is to favour darker colours (ideally black, $RGB = \langle 0.0, 0.0, 0.0 \rangle$) while avoiding lighter colours (such as white, $RGB = \langle 1.0, 1.0, 1.0 \rangle$). Li et al. used simulated annealing to find a suitable trade-off where the average RGB value of a pixel was kept low while maintaining contrasts between UI components. Using this approach they were able to produce a variant which, though less aesthetically pleasing, consumed less screen energy (40% on average). There are similar approaches to this such as dimming the screen [64], or simply inverting the colours [63], though such techniques suffer from a loss of colour relationships (i.e., properties like contrasts between UI elements, essential

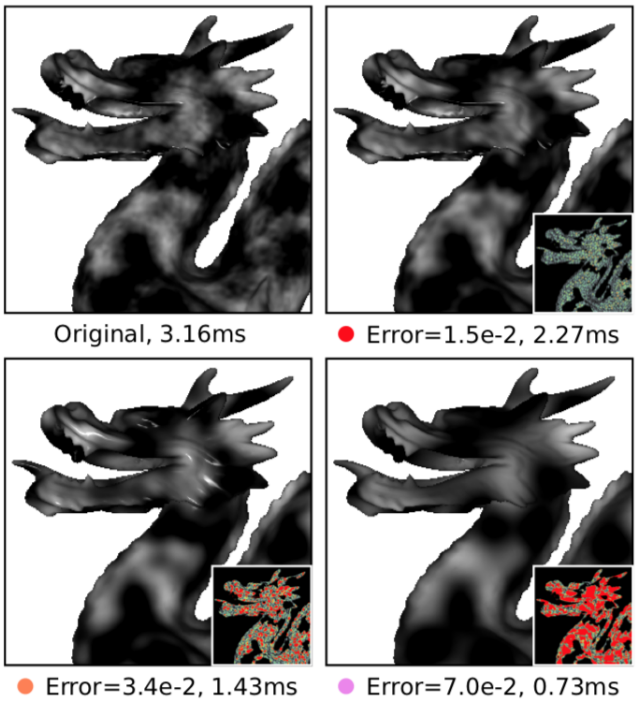


Figure 2.4: Approximated render outputs from Sitthi-Amorn et al's 2011 paper 'Genetic Programming for Shader Simplification' [158].

for ease of viewing).

Reducing energy consumption using approximation was also exploited in the development of PowerDial [92]. PowerDial toggles parameters of applications with the goal of reducing the execution time during periods of high use. The idea is that applications could run on servers and that during periods of low demand they would give more exact answers (which would take longer periods of time to process) while during periods of higher demand the outputs would be approximated (taking shorter periods of time to process). This would thereby reduce the necessity of having spare capacity on standby which would reduce the overall energy consumption of the computer system. Using approximate computing to reduce energy consumption is useful. The majority (80%) of software engineers who work in energy-constrained systems stated, in a 2016 study, they would be willing to sacrifice some functional requirements to reduce energy consumption [120].

While execution time has typically been the non-functional property of interest for software engineers, ecological and usability concerns have pushed developers into recognising the energy consumption of computer systems as an important concern. Though many, when considering the energy efficiency of a computer system, would point first towards the hardware it is important to keep in mind the role of software. Software's relationship with hardware is analogous to a driver's relationship with a car. If we view the car-driver relationship as a single system for traveling from A and B, vast differences in fuel consumption can be achieved depending on what route is taken, what speeds are maintained, and whether unnecessary engine revving is avoided. These are not decisions taken by the car. The car may have a stated efficiency (e.g. kilometres per litre of fuel) but this only serves as a factor in the overall efficiency of driving a car, the rest is determined by human decision making. For optimal efficiency in driving both the car and the driver must be optimal. To stretch this analogy one step further, there exists no 'optimal driver'. A driver can only be optimal for his given vehicle (for example, he knows what the optimal speed of his vehicle is which will not be the same across all models), in the same way a piece of software can only be optimal for a given hardware configuration.

It has been estimated that 1.9% of global CO_2 emissions in 2011 were due to information and computing technologies [36]; larger than the United Kingdom, which was estimated to have produced 1.47% of global CO_2 emissions during the 2010-2014 period [163]. Even if ecological concerns are to be ignored, the electricity generated (measured as between 1.1% and 1.5% in 2010 [102]) is costly and, as such, there are economic concerns also. Due to the emergence of smartphones and tablets, software engineers have to concern themselves with the usability constraints regarding energy consumption. There are now more smartphones in the world than personal computers [89], each of which has a limited energy budget which users expect to be utilised efficiently. It has been shown that 18.6% of applications in the Google Play (Android) market place, in which feedback exists, have complaints about perceived energy inefficiency of those applications [181]. However, it has been shown that software developers lack the necessary knowledge to make software more energy efficient [139], and metrics, previously believed to guide developers towards more energy efficiency solutions are, in reality, poor at doing so [150]. In response to this, GI researchers have started to target the area. They see that software can be refactored automatically to produce variants that are more energy efficient.

A good example of using genetic improvement to improve energy consumption is work by Manotas et al. [121] They improved the energy consumption of source code by optimising which `java.util.Collection` implementations were used; something which is understood to influence energy consumption [141]. They found that using their approach (which they called SEEDS — 'Software Engineer's Energy-optimization Decision Support framework'), they could reduce en-

ergy consumption of the applications targeted by up to 17%. Their approach exploited the fact that `java.util.Collection` implementations can be semantically equivalent and, thus, interchanged without alteration of semantic properties. For example, instances of `java.util.ArrayList` may be replaced with `java.util.LinkedList` without any error. Even in cases where semantics matter (e.g. a `java.util.HashSet` cannot always be replaced with an `java.util.ArrayList`), tests can be run to check semantics are preserved. While entities such as Sets or Lists are certainly not semantically equivalent, they may effectively be in certain contexts (say, in the context in which a developer wants a simple store of unique elements). Genetic improvement, with an adequate black-box test suite, is able to tease out these instances and optimise accordingly. A similar approach to Manotas et al. was adopted by Burles et al. to optimise Google Guava's `ImmutableMultimap` class (which contains many instantiations of `java.util.Collection` subclasses) [47]. They were capable of reducing energy consumption of the `ImmutableMulimap` class by 24%.

Noureddine et al. showed that simply changing certain design patterns could alter the energy efficiency of an application in a manner that guaranteed functional preservation [133]. Similarly, Banerjee et al. automatically refactored open-source Android applications and found that up to 29% decrease in energy consumption was achievable without breaking any functional properties [23]. Banerjee et al. did this by enforcing what they refered to as 'energy-efficiency guidelines': ensuring resources were acquired as late as possible and released as early as possible; removing resources acquire/release from nested structures; permitting some approximation if permitted; and rectifying instances where resources are acquired but never released.

Regardless of the functional property being targeted, we must remember that in all cases we are implicitly specialising software for a particular hardware setup. One of the major successes of modern software engineering is that of portability. Virtual machines, interpreters, and good compiler support have enabled software developers to code once and deploy almost anywhere. However, there is an inherent inefficiency in this: the same software may be deployed to many different hardware targets but it is unlikely to run optimally in all cases. For instance, in 2014 Langdon et al. [109] showed that optimising CUDA kernels for different GPU targets yielded different results and that results optimal for one target were not optimal for another. Fortunately, optimising to specific environments is a task genetic improvement of non-functional properties is good at. With a genetic improvement framework, a developer may code once, deploy everywhere, then have the code optimised for its deployment environment. In 2017 Yoo proposed embedding genetic improvement into programming languages [187] as a way of allowing this. He proposed using an annotation-based approach where developers would tag variables with `@optimize` to state they can be tuned, and `@approximate` to functions outputs of which may be approximated. These would then be tuned and approximated in the environments in which they were deployed. CLBlast [134], an OpenCL Basic Linear Algebra Subprograms (BLAS) library, does this to a limited extent. It differs from its main competition 'clBLAS', in that it incorporates an algorithm that automatically tunes its OpenCL parameters to the environment in which it is deployed. It has been shown that, using this very simple approach, CLBlast can improve upon clBLAS, its main competitor, achieving a factor of two speed-up in some instances.

Zhang et al. [188] introduced a novel idea that specialised hardware for a specific software target. They noted that directly mapped cache memory is more energy efficient per access than a set-associative cache, but only in cases where the cache hit-rate is low. The cache hit-rate is influenced by the cache size (which becomes more inefficient as size increases), and the application being run. Therefore, there is an 'optimal cache' for each application. They created a special cache

in which the size, and whether it is directly mapped, two-way, or four-way set associative, could be modified. Their idea was that these properties could be tuned (at the level of software) for the application to be run. They found that the average energy consumption by the cache could be reduced by an average of 40% after the cache was correctly tuned for a specific application.

Up until now we have discussed optimising non-functional properties at the level of source code. This is logical as it is the level in which a developer interacts and, therefore, the level in which they are most likely to understand, and accept, changes. However, there have been notable efforts to work at different levels. Schulte et al. optimised the PARSEC benchmark suite [31] by modifying the assembly code, reducing the energy consumed by up to 20%. Orlov and Sipper, created the 'FINCH' optimisation framework which targets Java byte-code [136, 137]. Their philosophy goes further and shuns the idea of attempting optimisation at the level of source code, stating in their 2011 work [137] :

> 'The source code is intended for humans to write and modify, and is thus abundant in syntactic constraints. This makes it very hard to produce offspring with enough variation to drive the evolutionary process.'

Their argument is reiterated throughout their work, highlighting that working at lower level code is unrestrictive in a manner that source code is not. Modern programming languages employ a variety of techniques to constrain developers from undertaking dangerous action but, Orlov and Sipper's idea is that this restricts avenues for optimisation.

2.3 Moving forward

In this Chapter we outlined the knowledge necessary to understand the remainder of the thesis. As has been discussed, meta-heuristics, in development for the last few decades, have found their way into software engineering. The software engineering community is just beginning to see what is possible using these techniques, in an area known as search-based software engineering, and their usage directly in the modification of software in an area known as genetic improvement.

As outlined in this thesis's introduction, we wish argue that genetic improvement is an avenue of research which may one day lead to the automatic improvement of software without human intervention. With particular emphasis on non-functional properties, we think the productivity of developers can be significantly improved if such techniques are further developed.

Prior to carrying out the research outlined in this thesis there were still many unanswered questions within the field of genetic improvement. Work on targeting non-functional properties outside of execution time was limited, leading to questions as to what *could* be optimised in a meaningful manner. Due to the emergence of energy-constrained mobile devices, optimisation of energy consumption via genetic improvement seemed to be a fruitful avenue for research, though prior to our investigations, optimisation of energy consumption had primarily been carried out with estimates of energy consumption on GI frameworks not representative of the state-of-the-art.

At the time of carrying out our research, we would have classified the state-of-the-art in non-functional GI to have been Langdon and Harman's work on Bowtie2 [108], given the impressive performance gains achieved on an application of non-trivial size. However, the general applicability of this approach was poorly understood. We show, in Chapter 3, that the approach is capable of optimising the energy consumption of a small application (MiniSAT) but fails to scale to optimise four larger applications (7zip, Bodytrack, Ferret, and OMXPlayer). Though we still think the

work by Langdon and Harman is impressive, it does not appear to be generalisable. The *delete*, *copy*, and *replace* operators which Langdon and Harman used, produce a search space which was not properly understood, and, motivated by our discovery that this approach failed to optimise larger applications, we carried out an investigation into the search space these operators produce (discussed later in the latter half Chapter 3) under different conditions. Of particular interest to us, we found that approximation of output was a good avenue for optimisation of non-functional properties.

Wu et al.'s work on deep parameter optimisation [184] had previously shown us that non-functional properties could be traded off with one another — which we considered to be state-of-the-art in multi-objective GI research. We therefore set out to test deep parameter optimisation in the field of approximate computing, where functional and non-functional properties are traded off with one another. This investigation is discussed in Chapter 4, first focusing on trading-off functional correctness against execution time, then against the energy consumption of an Android application running on a mobile device.

From this interest in optimising energy consumption on mobile devices, we decided to focus on optimisation of software running on modern heterogeneous architectures. Prior to this work, there had been some work in GI to optimise CUDA kernels [109] (code designed to be compiled and run on nVidia GPUs), but very little on automatically translating sequential code to run on GPUs. This was an avenue of research we believed to be a good starting point for the introduction of genetic improvement into the field of heterogeneous hardware optimisation. We discuss our research into this area in Chapter 5. We finally conclude with Chapter 6 to summarise what we have discovered. It is hoped this thesis will inspire others to expand upon what has been discussed, and help the software engineering community develop new tools and techniques to tackle the problems they face.

Chapter 3

Optimising software's energy efficiency

In this chapter we report on investigations into using genetic improvement to optimise software's energy consumption. The research outlined in this chapter started in 2014. At this time genetic improvement had largely been utilised for the optimisation of functional properties (bug fixing, primarily) and execution time, with optimisation of other properties, such as energy consumption, more hypothesised as a viable target [86] rather than properly explored.

This chapter outlines our journey into exploring energy consumption as a viable target for genetic improvement. The major sections of this chapter highlight three pieces of work, presented in chronological order. The first, is an initial investigation into the matter (Section 3.1), which details using GI to improve energy efficiency on a small program. While somewhat successful, it was decided more data was required to draw more meaningful conclusions, thus leading us to the research detailed in Section 3.2. The results of this section indicated our approach to GI did not scale as well as we had anticipated. This spurred an investigation into the nature of energy search spaces, described in Section 3.3.

3.1 Initial Investigation

One of the largest hurdles in producing energy-efficient software is the developer's disconnect between the source code they write and the energy that will be consumed from the compiled product they deliver [121]. Without a deep understanding of how a particular compiler works, along with an equally deep understanding of how much energy a given machine code instruction will consume, the problem remains difficult for developers. Subtle changes, such as introducing inline methods [157], swapping API implementations [121], and constructing semantically equivalent (but structurally inequivalent) algorithms [46] have all been shown to influence software's energy consumption. However this influence is difficult to determine outside of the ad hoc and inefficient process of trial-and-error. Tools have been developed to highlight energy-inefficient areas of software [22, 56, 115, 82] though developers retain responsibility for rectifying these inefficiencies.

We suggest that the most under explored method of decreasing software's energy consumption lies in automated processes. Such processes would allow developers to focus solely on meeting

the software's functional requirements with worries about energy consumption left to an algorithm capable of refactoring software to a more optimal state. This is a natural target for genetic improvement.

With this idea in mind, we attempted to use currently available GI tools to optimise software to improve its energy efficiency. At the time of carrying out this investigation, there was very little consensus on the best way to carry out such research. We therefore chose to undertake a small investigation, using what basic techniques and tools were available to optimise the energy efficiently of a small, easily testable application.

We targeted MiniSAT, a popular Boolean satisfiability solver (SAT Solver). In 2014 Petke et al. demonstrated that MiniSAT could be optimised for execution time, via a simple genetic improvement approach, to produce a solver that was 17% faster than any human-written equivalent in the Combinatorial Interaction Testing (CIT) domain [145]. SAT solvers may be optimised for a variety of downstream applications such as AI planning [100] through to package management [166] and predicting crosstalk in integrated circuits [51]. With this in mind we took MiniSAT and, using a GI setup influenced by that used by Petke et al., and attempted to optimise the SAT solver's energy efficiency while specialising for three different downstream applications — CIT, Ensemble Computation, and AProVE [1].

We set out to answer the following research questions:

RQ1 To what extent can MiniSAT's energy consumption be reduced using genetic improvement?

RQ2 Do different downstream MiniSAT applications require different optimisations?

RQ3 Does reduction in energy consumption correlate to reduction in execution time when GI is applied?

3.1.1 Overview of our genetic improvement algorithm

We built on an approach to GI first outlined by Langdon and Harman [108] for their work on Bowtie2 and later modified by Petke et al. in 2013 [144] and again in 2014 [145] for their MiniSAT experiments. We modified the fitness function to take into account estimates of MiniSAT's energy consumption made by the Intel Power Gadget (see Section 3.1.5) instead of the number of lines executed (a metric previously used to optimise for execution time). We also modified the selection and mutation procedure in an attempt to tackle the phenomenon of bloat; a problem in Evolutionary Computation where solutions grow in size without any improvement in the fitness [24]. As in Petke et al.'s work [144, 145], we targeted MiniSAT's `Solver.c` class; the class which contains the main SAT solving algorithm.

3.1.2 Program and Genotype Representation

In line with the techniques used by Petke et al. [145] when optimising MiniSAT for execution time, we made modifications at the level of source code lines. We used three different GI operators: *delete*, *replace* and *copy*. A *delete* operation removed a line, a *replace* operation replaced one line with another, and a *copy* operation copied a line to another location. Though other experimenters have had some success with incorporating external lines of code [145], we only used the genetic material

[1] Petke et al. later specialised MiniSAT for these downstream applications (to reduce execution time) in a 2017 journal extension of their 2014 work [143].

present within the targeted source file (`Solver.c`), due to the observation that code required to provide an software varient is often contained somewhere within the target codebase (i.e, the Plastic Surgery Hypothesis [26]).

We applied these *delete*, *replace*, and *copy* operators to a tagged representation of source code; a representation specially created for GI research, first introduced by Langdon and Harman in 2010 [107] and later utilised in a variety of other GI investigations [108, 109, 145]. We refer to this as the *Langdon format*, an example of which is shown in Figure 3.1.

```
<Solver_235>      ::=    " if"<IF_Solver_235> " \n"
<IF_Solver_235>   ::=    "(order_heap.empty())"
<Solver_236>      ::=    "{\n"
<Solver_237>      ::=    "" <_Solver_237> "\n"
<_Solver_237>     ::=    "next = (-1);"
<Solver_238>      ::=    " break;\n"
<Solver_239>      ::=    "}\n"
```

Figure 3.1: A `Solver.c` snippet converted to the Langdon format.

To translate code to the Langdon format each line is labelled with a unique identifier. These identifiers indicate whether a line is modifiable or not. If a line is not modifiable its identifier follows the format of `<FILE_ID>`. In this work, opening and closing curly brackets, variable initialisations, and function declarations, were tagged as unmodifiable. In the case of `IF`, `WHILE`, and `FOR`, only the conditions, and the pre- and post-statements in the case of `FOR`, can be modified. In Figure 3.1, `"(order_heap.empty())"`, and `"next = (-1);"` are modifiable.

As the operators to be applied function on source code lines, the code to be tagged was formatted so that each statement was on its own separate line to avoid modifications being applied to multiple statements. Opening and closing curly brackets were moved to separate lines so any modifications to lines containing a statement did not interfere with program scopes. As MiniSAT is written in C, a language which permits bracketless one-statement `FOR`/`WHILE`/`IF` structures, we refactored any instances of these to be enclosed in curly brackets. This was done to reduce errors that may arise from deleting the bodies of these bracketless structures.

Once converted to the Langdon format the source code could be modified by simply *deleting*, *replacing*, or *copying* a line identifier (e.g. `<_Solver_237>`), taking into account the aforementioned restrictions. The format could then be converted into the original source code by taking the unmodifiable lines and expanding them. For example, in Figure 3.1, `<Solver_237>`, an unmodifiable line, references `<_Solver_237>`. When converted back to source code it is expanded to produce `next = (-1);`. If a *delete* operation were to be applied to `<_Solver_237>` then the rule for that identifier would read `<_Solver_237> ::= ""` and `<Solver_237>` would simply expand to an empty line. Special cases existed for conditional statements: predicate expressions could only be replaced with predicates found in the same type of conditional statement, the *delete* operation replaced a conditional with '0' to avoid compilation issues, and *copy* operations were not permitted on conditional statements. For example, a `WHILE` loop condition (e.g., `x>5`) could only be replaced with the condition of another while loop (e.g., `y==2`) and could never be replaced with a random statement found elsewhere in the software (e.g., an assignment such as `z=z*2;`). If a *delete* operation were to be applied to the `WHILE` loop, it would read `while(0)`; a by-proxy

removal of that branch from the code.

Figure 3.2 shows how these *delete*, *copy*, and *replace* modifications are represented inside a population of solutions. In this work, a genotype is a list of modifications made to the Langdon format which was then translated to its phenotype (the source code) by expansion.

```
#delete line 205
<_Solver_205>

#replace if condition in line 154 with if condition
#in line 307
<IF_Solver_154><IF_Solver_307>

#copy line 299 and insert above line 325
<_Solver_325>+<_Solver_299>

#remove line 56 and replace line 78 with line 145
<_Solver_56> <_Solver_78><_Solver_145>
```

Figure 3.2: An example population of four solutions using the Langdon format.

A modifiable identifier alone, `<_FILE_ID>`, is a *deletetion*; a modifiable identifier followed, without a space, by another, `<_FILE_ID><_FILE_ID>` is a *replace* operation that replaces the former with the latter; and two identifiers separated by `+`, `<_FILE_ID>+<_FILE_ID>`, is a `copy` operation that copies one line (the latter) to another location in the source code (above the former). A space separates multiple operations.

3.1.3 Fitness Function, Selection, Crossover and Mutation

Algorithm 3 shows the genetic improvement algorithm we used in this study. The algorithm starts with a population of size S, created by iteratively adding individuals of only a single modification (i.e. a single *deletion*, *replace*, or *copy* operation), selected uniformly from the set of all possible modifications.

The population then iterates through a number of generations, G. Each generation consists of producing a new population P' that replaces the former population P. A generation starts by populating P' with individuals from the previous population P. This process involves iterating through the population P and inserting all individuals into population P' which, after evaluation, are said to have passed, are in the top half of the population P in terms of fitness, and have a fitness greater than 0.95. In this setup an evaluation consists of measuring the energy consumed by the solution to-be-evaluated (see Section 3.1.5 for more details on how this is achieved) across all tests selected from the testing set (test selection is explained in Section 3.1.4). To normalise, the energy consumed by the original unmodified software for those tests is divided by the energy consumed by the solution for those tests. Thus, the fitness of a solution tends higher as the solution's energy consumption decreases and a fitness of 1 means there has been no change.

As all tests in our setup are Boolean satisfiability problems, a test was deemed to have passed when the modified solution categorised that test as satisfiable or unsatisfiable (i.e. without crashing or producing no output), with this categorisation equal to that produced by the original application.

Algorithm 3 The Genetic Improvement algorithm used when modifying MiniSAT.

Require: G, the number of generations, $G \in \mathbb{N}$
S, the population size, $S \in \mathbb{N}$
M, the set of all possible modifications
$top(P, N)$, returns a sorted vector of the top N solutions (based on fitness) from pop P
$crossover(p_1, p_2)$, returns a child from two parents, p_1 and p_2
$random()$, returns a random number, r, where $r \in \mathbb{R} \wedge 0 \leq r \leq 1$
$uniformSelectSolution(P)$, returns a uniformly selected solution $p \in P$
$uniformSelectMutation(M)$, returns a uniformly selected modification, $m \in M$

$P \leftarrow \{\}$ # $p \in P : p$ is a vector of modifications $m \in M$.
while $|P| < S$ **do**
  $p \leftarrow \langle unformSelectMutation(M) \rangle$
  $P \leftarrow P \cup \{p\}$
end while
for $1..G$ **do**
  $P' \leftarrow \{\}$
  for $p \in P$ **do**
    if $p.passed(p) \wedge fitness(p) > 0.95 \wedge p \in top(P, (|P|/2))$ **then**
      $P' \leftarrow P' \cup \{p\}$
    end if
  end for
  $P'_c \leftarrow \{\}$
  $i \leftarrow 0$
  $T \leftarrow top(P', |P'|)$ # E.g., T is P' sorted by fitness (descending), $T = \langle p_0, p_1, ..., p_{|P'|-1} \rangle$
  while $|P'| \neq |P'_c|$ **do**
    $j \leftarrow i \mod |P'|$
    $p_x \leftarrow T_j$
    $p_y \leftarrow uniformSelectSolution(P')$
    $P'_c \leftarrow P'_c \cup \{crossover(p_x, p_y)\}$
    $i \leftarrow i + 1$
  end while
  for $p' \in P'$ **do**
    if $p' \notin top(P, 0.05S) \wedge random() < 0.5$ **then**
      $p' \leftarrow p'^{\frown} \langle uniformSelectMutation(M) \rangle$
    end if
  end for
  $P' \leftarrow P' \cup P'_c$
  while $|P'| < S$ **do**
    $p \leftarrow \langle uniformSelectMutation(M) \rangle$
    $P' \leftarrow P' \cup \{p\}$
  end while
  $P \leftarrow P'$
end for

In our GI algorithm, a small degradation in energy efficiency is permitted to allow the algorithm to escape local optimal in the search space.

Next a new set, P'_c, is created which is used to temporarily store the children of those selected individuals in P'. To create these children, crossover is carried out between two parents, $p_x \in P'$ and $p_y \in P'$. p_x is chosen, by selecting the top Xth fittest solution from the population, where $X = i \mod |P'|$, and i is an integer which increments for each child generated within the current generation. p_y is selected uniformly from population P'. Crossover is then carried out which, in our setup, simply means appending the modifications of the first parent onto the second. This child is then added to P'_c This process repeats while $|P'| \neq |P'_c|$ (i.e. the algorithm creates as many children as there are solutions selected from P').

Next, for each solution in P', mutation is carried out with a probability of 0.5 if the solution is not in the top 5% of the population (i.e., we enforce some elitism). Our mutation operation simply consists of appending a single random modification to the solution. After this we add P'_c to P'.

As a final step, if P' does not meet the population size, the algorithm adds solutions consisting of a random single modifications to the population (in the same manner as was done when generating the initial population) until the population size is met. The population P is then replaced by the new population P'. The algorithm ceases execution after G generations.

3.1.4 Applications targeted — Training and Test data

For each MiniSAT downstream application a training and test set were constructed, representative of that application's use. Within each generation only a small subset of the training set was selected to evaluate the solutions. A subset was chosen to avoid over-fitting and to reduce execution time. The number of tests was dependent on the MiniSAT application: four for CIT and five for Ensemble and AProVE. To ensure this selection was representative of the entire training set, a binning system was introduced (similar to that used by Langdon and Harman [108] in their work optimising Bowtie2 which decomposed the training set tests into bins based on complexity and type). A single test was uniformly selected from each bin in each generation. For our experiments we formed bins based on execution time and Boolean satisfiability. Bins 1 and 3 contained satisfiable solutions while bins 2 and 4 contain unsatisfiable solutions. Bins 1 and 2 contained tests with small execution times while bins 3 and 4 contained tests with larger execution times. The 5th bin, if present, contained both satisfiable and unsatisfiable solutions that had a larger execution time than those in bins 3 and 4. This test case selection process guaranteed that fitness evaluation was always carried out against both satisfiable and unsatisfiable tests of varying difficulty.

The following subsections describe the MiniSAT downstream applications we specialised for and the composition of the training and test sets used. We carefully selected three applications areas we believe are suitably diverse in real-world or academic usage. For each we describe the purpose of the downstream application and give a description of the training and test set provided. It should be noted that the test cases we used run in a considerably smaller execution time than what would normally use benchmark MiniSAT. This was to keep evaluation times low, thereby permitting more evaluations to be carried out in our limited time budget.

Specialising for CIT

Combinatorial Interaction Testing (CIT) is a black box test sampling technique used to test highly configurable software [132]. With highly configurable software, such as database management systems and architectures like software product lines, testing all configuration combinations is infeasible

though it remains important to ensure no combination of configuration variables exist that may result in software failure. CIT's role is to produce a test suite which sufficiently covers the configurations while minimising the execution time of such tests. CIT has been successfully translated and run as a Boolean satisfiability problem [21, 132], though running these SAT problems is a computationally intensive task.

For this investigation we created a training set that contained 58 tests, 23 of which were satisfiable, spread over the 4 bins. The mean execution time for bins 1 and 2 was 3.33s, and the mean execution time for bins 3 and 4 was 10.68s. The test set contained 20 tests, 11 of which were satisfiable with an average execution time of 13.07s.

Specialising for Ensemble Computation

Ensemble Computation is the study of an NP-complete variant of the Boolean circuit problem where the objective is to find the smallest circuit that satisfies a set of Boolean functions simultaneously [95]. This problem can be translated into a satisfiability problem, which MiniSAT can then seek to solve.

Our Ensemble training set contained 25 tests, 12 of which were satisfiable, spread over 5 bins. The mean execution time for bins 1 and 2 was 4.21s, and the mean execution time for bins 3 and 4 was 9.89s. The average execution time for bin 5 was 28.87s. The test set contained 14 tests, 4 of which were satisfiable, with an average execution time of 14.23s.

Specialising for AProVE

AProVE, Automated Program Verification Environment [72, 73], is a system for the generation of automated termination proofs of term rewrite systems. AProVE uses a Boolean satisfiability solver to determine which paths within a program can or cannot be reached. Proving termination is a much discussed area in computer science [37, 59, 71, 117] with SAT solvers frequently used to aid analysis [53, 153]. In this example, the SAT solver is a component in a much larger application. This distinguishes it from the two other applications we optimise where SAT solvers take a central role in problem solving.

The AProVE training set contained 24 tests, 13 of which were satisfiable, spread evenly over 5 bins. The mean execution time for bins 1 and 2 was 6.27s while the mean execution time for bins 3 and 4 was 19.04s. The average execution time for bin 5 was 25.33s. The test set contained 11 tests, 5 of which were satisfiable, with an average execution time of 17.83s.

3.1.5 Estimating Energy Consumption

We estimated the energy consumed in computing these tests using the Intel Power Gadget API [186] for Mac OS X which estimates the energy consumption of 2nd Generation and higher Intel Core processors. Given MiniSAT's singled-threaded, CPU-bound nature we believed this method of estimation was suitable for our requirements, however, it should be noted that Intel Power Gadget estimations do not include energy consumed within other hardware components.

Intel Power Gadget uses drivers and libraries to read the processor's special energy model-specific registers (MSRs) over a specified time period. These register readings are then used to calculate the total energy consumed.

We built the Intel Power Gadget API into a C++ application that took a command-line command as input and estimated the CPU's energy usage during that command's execution. Running

MiniSAT against a test case using this program gave us the energy consumption of MiniSAT for that test case.

When carrying out our experiments we were careful to avoid running processes that may cause large variations in energy estimations. All unnecessary background processes were terminated while experiments were run. Running the original MiniSAT on a typical test case for 200 iterations, taking energy readings each time, we found a standard deviation of 5.23% in estimates; a variance within the limits we deemed acceptable for our experiments.

3.1.6 Experiment Setup

We targeted the main solving algorithm in MiniSAT2-070721 (hereinafter simply referred to as 'MiniSAT') for modification in our GI framework. For each downstream application, the GI framework was run for 20 generations with a population of 100. Once complete any solution generated that achieved a fitness greater than 1.05 was run against all tests within the training set to give an overall ranking of the best solutions based on the total energy consumed. This step was included as it was found that many solutions with a high fitness only performed well on the tests selected for that generation. It can be considered a method of determining the 'true fitness' of the best solutions, as opposed to the fitness value given by the GI framework. The best solution was chosen from this 'true fitness' list and declared the 'champion' solution.

The champion solution was then run against the entire test set for its downstream application 20 times with the energy readings averaged to give a value which was then compared to the original MiniSAT's performance against that test set (again, run 20 times then averaged) to obtain an overall percentage improvement.

We ran the GI algorithm three times, each time specialising MiniSAT for a different downstream application. In practice each downstream application simply required the use of a different training set (and test set for the final results). After gathering data from these three runs of the algorithm, we answered our research questions.

To answer **RQ1** (*To what extent can MiniSAT's energy consumption be reduced using Genetic Improvement?*) we observed the energy improvement between the original, unmodified MiniSAT and the champion for each downstream application. As the champion solution and the original MiniSAT were run 20 times on the test set we were able to determine how statistically significant these results were.

RQ2 (*Do different downstream MiniSAT applications require different optimisations?*) was answered by running the champion solution for each downstream application against the test sets of the other applications. If a comparable energy reduction was observed when using other test sets then we could argue the modifications applied are general (in the sense they are improvements across all the downstream applications tested), however if the energy reductions were significantly smaller, crashes occurred, or timeout events were triggered then we could conclude modifications must be specialised in some manner. We also analysed the modifications made to the software to produce the champion solutions in an attempt to determine whether any similarities could be found. Where solutions were found to be specialised, we investigated why modifications applied to one champion were less effective (or ineffective) when used within another MiniSAT domain.

To answer **RQ3** (*Does reduction in energy consumption correlate to reduction in execution time when GI is applied?*), we took each experiment champion and measured the total time required to compute all tests within their respective test set. These times were then compared to the execution time of the original, unmodified software when computing the test set to give a percentage reduction

in execution time. We then compared these measurements to the energy percentage improvement. The purpose of this was to see if energy reductions corresponded to reductions in execution time. To obtain a deeper understanding of the time-energy relationship we also sampled random solutions from each run of the algorithm (20 from each experiment, 60 in total) and measured their execution times and energy consumption for each downstream application, on their respective test sets. Analysing the data, we were able to determine how correlated these readings are.

3.1.7 Results and Discussion

Here we report the results obtained from carrying out the experiments described in Section 3.1.6.

RQ1: Energy Reduction

Application	Original (J)	Original SD	Champ (J)	Champ SD	Reduction (%)
CIT	3111	15 (0.5%)	2969	15 (0.5%)	4.58
Ensemble	2232	12 (0.6%)	1665	10 (0.6%)	25.39
AProVE	3145	35 (1.1%)	2973	35 (1.2%)	5.44

Table 3.1: The original MiniSAT's total energy consumption across all test set tests compared against the champion solutions' energy consumption.

All three experiments produced champion solutions which out-performed the original MiniSAT, in terms of energy efficiency, for their respective test sets. Table 3.1 shows the improvements. The CIT and AProVE champions achieved modest energy consumption reductions of approximately 5% while the Ensemble champion achieved a reduction of 25.39%. Further analysis of the results showed that these energy estimations are statistically significant ($p < 0.01$) using the Wilcoxon signed rank test.

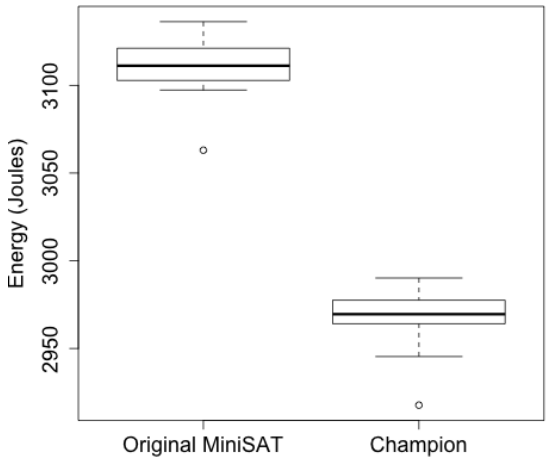
We carried out the Vargha-Delaney-A statistic for the results of all three runs of the GI algorithm to determine effect size. We found each to have a score of 1. This score shows that the energy efficiency of the champion solutions are superior to the original MiniSAT in all cases, for their respective application domains. Figure 3.3 shows boxplots that visually demonstrate these effect sizes.

When taking into account that the `Solver.c` code targeted is relatively small (478 lines of code), and that it is already considered to be quite efficient (at least in terms of execution time), we found it encouraging that a reduction in energy consumption of 25% was achieved.

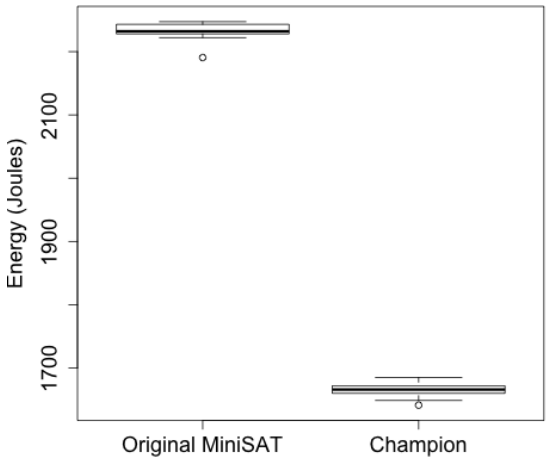
RQ2: Specialisation differences

-	On CIT	On Ensemble	On AProVE
CIT	-	X	X
Ensemble	X	-	X
AProVE	3.56%	3.86%	-

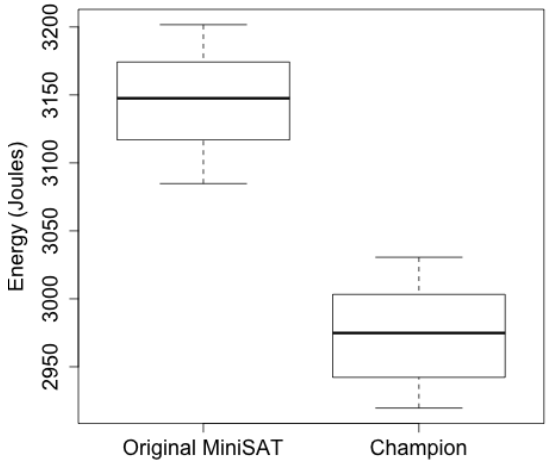
Table 3.2: The best solutions' energy consumption when computing other test sets.



(a) CIT



(b) Ensemble



(c) AProVE

Figure 3.3: Boxplots of the champion solutions' energy consumption compared to that of the original MiniSAT.

When the champion solution for each application was run against the test sets for the other two (see Table 3.2) we found that both the CIT and Ensemble champions timed out when attempting to run on the other two test sets (the timeout was set to 5 minutes per test case, and no test set exceeded 90 seconds when run on the original MiniSAT). This indicates that these champions are 'specialised' in a way in which they cannot be generalised as optimisations for all SAT problem sets. The AProVE champion functions correctly on the other test sets but does not achieve the same performance improvement as seen when run in AProVE domain. It appears that this may be a general improvement to MiniSAT unlike the other two solutions.

The AProVE champion solution was found to be the removal of a complex assert statement. Removing assert statements has previously been shown to produce good results for execution time when optimising MiniSAT [145]. Our experiments show that the same is true for optimising energy consumption. It is also easy to understand why such a modification does not result in specialisation as it will not produce a version of MiniSAT that differs semantically to the original, unmodified application.

The more interesting results come from the specialisation cases (CIT and Ensemble). The CIT solution was found to be a single modification disabling an `IF` statement (the condition replaced with a zero through the *delete* operation) in MiniSAT's `pickBranch` function. The statement is used for picking a random variable for assignment and is called 2% of the time. The condition within the `IF` statement itself involves running a random number generator which may be an unnecessary cost for such an under-used piece of code. It is currently unknown why this modification results in the solutions performing so poorly on the AProVE and Ensemble test sets. It is perhaps the case that this rarely entered `IF` statement does result in significant impacts on performance for Ensemble and AProVE, to the extent that it is of benefit for them, but not for CIT.

For the Ensemble application, the application in which we achieved the largest reduction in energy, we found the specialisation made a single modification to a `switch` statement. A modification equivalent in outcome to changing MiniSAT's polarity mode from `polarity_false` to `polarity_true`. This caused the solver to try an assignment of `True` instead of `False` to each variable that had been picked for branching. It seems that changing the polarity mode from false to true is of benefit in Ensemble Computation, though, as previously stated, why these changes produce measurable gains in performance is not fully understood.

One of the more unexpected, but nonetheless interesting, observations is that the champion in each run of the GI algorithm was always a genotype containing only one modification. We reject the idea that single modifications are truly optimal. Not only does this run counter to optimal solutions found in similar experiments [108, 145], the champion solution found within the AProVE experiment is a general optimisation which could easily be appended to the champions found for CIT and Ensemble to produce better results for each. Though mutation, and to a lesser extent crossover, always carries a high risk of producing poor solutions, it may be the case that our policy of elitism and employing a mutation rate of only 50%, made it more difficult to produce longer, potentially fitter genotypes.

RQ3: Energy-Time Relationship

Table 3.3 shows the reduction in execution time for each specialisation. When compared to the energy reduction results in Table 3.1 it can be seen that the values appear to be correlated.

To investigate the correlation further we uniformly sampled 20 functionally correct solutions from each GI run and executed each solution against their respective test sets with both energy and

Application	Unmodified(s)	Champion(s)	Reduction
CIT	268	261	2.58%
Ensemble	219	162	25.89%
AProVE	280	261	6.69%

Table 3.3: The original MiniSAT's execution time across all test set tests compared to the champion solutions' execution times.

time measured for each test. Figure 3.4 shows the data produced from this analysis with each test case's energy and time costs plotted. Visually the relationship between energy and time is stark, Table 3.4 gives the Pearson Correlation Coefficient for each experiment showing each to have a very strong energy-time correlation.

Application	N	Correlation	p
CIT	1160	0.988	$\ll 0.01$
Ensemble	457	0.995	$\ll 0.01$
AProVE	481	0.989	$\ll 0.01$

Table 3.4: The number of data-points, the Pearson Correlation Coefficient and p-value for the energy-time relationship in each experiment.

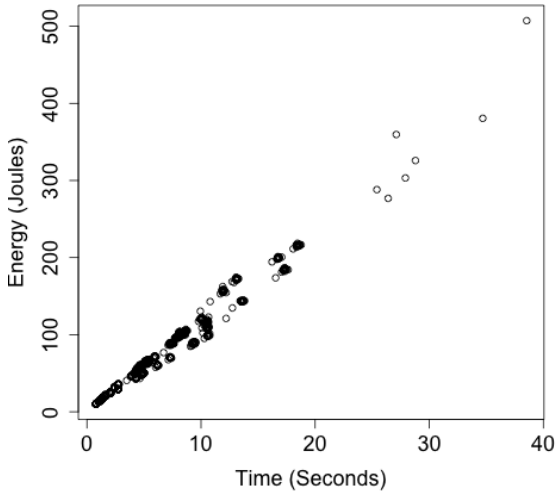
These findings show that for CPU-bound processes, such as MiniSAT, optimising execution time exclusively has the side effect of producing more energy-efficient solutions (and vice-versa). This provides further incentive to investigate using execution time as a metric for reducing energy consumption for CPU-bound, single-threaded applications. If energy consumption and execution time can be bundled into a single metric, developers may find it easier to reduce energy as methods to reduce program execution time are already well understood.

Although this result is somewhat expected and unsurprising, the effect has not been demonstrated empirically in the context of SBSE. We are therefore pleased this work may aid future research in providing evidence to their claims about energy-time relationships in CPU-bound, single-threaded applications.

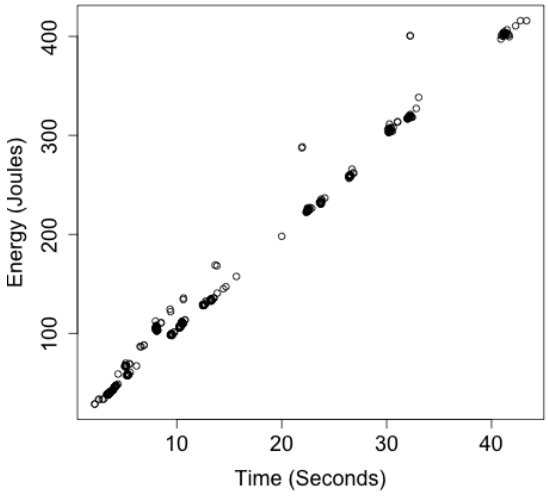
3.1.8 Threats to Validity

It is worthwhile mentioning there exists some threats to validity for the results produced and the conclusions drawn within this investigation. The first is the relatively small area of code optimised (478 lines). Working on such a small example is essentially reducing the search space, thereby making it easier to navigate and optimise for. Techniques have been introduced for GI to optimise larger applications [108] but they have only been demonstrated for execution time optimisation and automatic bug fixing [127]. This investigation does not tell us how such techniques would translate over to optimising for energy consumption.

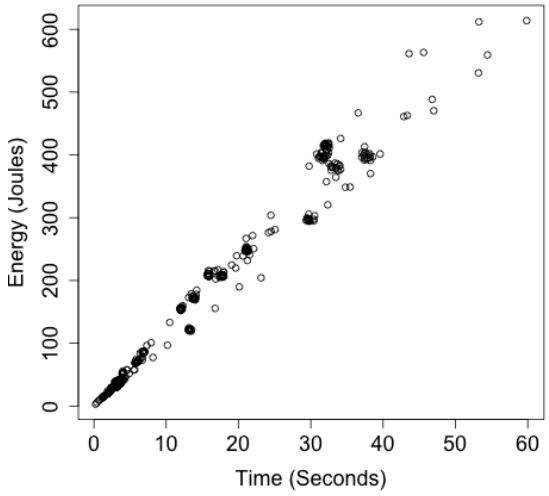
The Intel Power Gadget was chosen as a power estimation API for this investigation as it produced estimation with low variance, was easy to implement, and is supplied by a reputable hardware manufacturer. However, after this investigation, question still remained over how accurate this API is. Intel provide no formal documentation on this API, nor has there been any research into its validity meaning this research has been undertaken on the assumption the Intel Power



(a) CIT



(b) Ensemble



(c) AProVE

Figure 3.4: Scatterplot of the energy-time relationship within the three MiniSAT experiments.

Gadget functions within an acceptable degree of accuracy. Furthermore, the gadget is limited to CPU activity exclusively, and, therefore, results presented here ignore any potential energy increases occurring elsewhere and limits the findings to CPU-bound applications.

3.1.9 Summary

Here we documented an investigation where we evolved MiniSAT, a popular Boolean satisfiability solver, to reduce its energy consumption by specialising for three different downstream applications. We found that energy efficiency could be improved by as much as 25%, though it varies greatly depending on the downstream application being optimised. We also discovered GI is able to find solutions to reduce energy consumption that would be difficult for human developers to find.

Two of the three champion solutions were found to be specialised to their respective downstream applications to the extent they were no longer applicable to other MiniSAT downstream applications. This finding adds to the arguments presented in previous research that GI can be used to specialise solutions for specific environments [108, 145].

Our further investigations into the results produced in our experiments found that the energy savings corresponded to decreases in execution time most likely due to the CPU-bound nature of the application optimised. The very strong correlations found provide evidence that for applications with similar characteristics to MiniSAT (that is CPU bound, singled threaded, and with limited I/O activity) execution time and energy consumption may be considered interchangeable metrics.

3.2 Optimising Larger Applications

In the previous section we outlined an initial investigation in using genetic improvement to optimise software's energy efficiency. We highlighted some of the investigation's threats to the validity, primarily that we focused on a single, small target (`Solver.c` in MiniSAT, which only contains 478 lines of source code). Another threat was our use of an estimation of CPU energy consumption (via the Intel Power Gadget [186]) rather than direct measurements of the entire computer system.

The investigation outlined in this section was what we thought of as the 'logical followup' of the previous investigation. Our idea was to carry out a larger investigation, with more software targets, of non-trivial size, using direct energy measurements.

Though considerable effort went into setting up the framework necessary to run these experiments, we concluded that, ultimately, the approach to GI outlined in the previous investigation did not lead to significant improvements on larger applications. We report the setup used, and the (partial) results obtained. We present this section as part of this chapter as it serves as a bridge between the initial investigation and the investigation highlighted in Section 3.3. In line with good scientific practise, we present these null results here so others may learn from them and avoid such mistakes in the future.

3.2.1 Measurement Framework

As already mentioned, one of the biggest weaknesses with our initial investigation into optimising software's energy efficiency was the manner in which we determined the energy consumption of the

target application. The challenge of measuring energy is one of engineering. In theory, the setup we desire to carry out energy-centric GI research is simple: take a program (modified or otherwise) along with an input and measure its energy consumption during execution. In practise, however, this is not so simple. One must choose between direct and indirect measurements, and contend with the cost of taking measurements, since program execution can be costly.

Most previous search-based approaches to optimising the energy efficiency of software have estimated energy consumption [43, 47, 179]. These estimates can miss important high or low energy events thereby directing search away from an optimal solution. We therefore set out to develop a framework which makes direct energy measurements rather than relying on either estimates or simulations.

Programs are one component of a larger system: the computer that executes them. At present, one cannot directly measure the energy consumption of a program, because existing devices do not expose the coupling points between hardware components or operating system processes. In our previous investigation, we were able to *estimate* the energy consumed by the system's CPU but we could not ensure there were not significant energy events occurring elsewhere in the system. Our solution was to directly measure the energy consumption of the entire computer system. Measuring the whole system carries with it the challenge of contending with statistical measurement error due to events external to the program, like OS background processes. We mitigated these effects by taking multiple measurements and averaging the results.

Directly measuring the energy consumption of a program entails running and, if we were to, as we have just argued, run solutions multiple times to reduce noise, it follows that our framework must be efficient and scalable so we are not crippled by the costs of solution evaluation. For instance, a GI algorithm run with a population of 100, over 10 generations may require up to 1000 evaluations. If each evaluation costs several minutes (which is not uncommon when verifying the correctness of software), then it is easy to see how scalability becomes an issue. Especially as, ideally, we would want to permit many more evaluations than this. Fortunately, the process of evaluating of solutions is easily parallelisable, and the framework we developed exploits this fact.

We created a cluster of individual computer systems, each of which can measure their own energy consumption. Jobs (programs, modified or otherwise, along with input data) are sent from a client to the cluster's master node, which then distributes jobs to nodes with a maximum of one job running on any given node at any time. This node then measures its energy consumption while running the job. The energy measurement, along with the output of the job, is then returned to the client via the master node. Figure 3.5 shows our framework's layout with 2 nodes.

The nodes in this cluster are Raspberry Pi 2 Model B devices [9], each running Raspbian OS [10], a GNU/Linux OS based on Debian. The Raspberry Pis were chosen as they provide a cheap computer, representative of a real-world system in terms of architecture and their running of a Unix-based operating system. Each Raspberry Pi node can measure its own energy consumption via a MAGEEC Energy Measurement Board [5].

MAGEEC Energy Measurement boards are simple, inexpensive devices which sample the voltage drop across a resistor inline to the target's power supply (the Raspberry Pis' power cable in our case) at a sustained rate of 2MHz. A micro-controller on the MAGEEC Energy Measurement board listens for start and stop commands over a USB connection. When a start command is received the micro-controller begins sampling measurements and sends readings across the USB connection until a stop command is received. The MAGEEC board is controlled by a separate Raspberry Pi device (we refer to this as the 'measurement board controller') that is responsible for issuing the start and stop requests and reading energy data from the USB connection. As all the Raspberry Pi

devices (nodes and measurement board controllers) are on the same network, nodes send requests to their respective measurement board controller to start, stop, and receive energy measurements. Each MAGEEC board can measure up to three targets at once and, therefore, the ratio is three Raspberry Pi nodes to every MAGEEC board plus an additional Raspberry Pi (the measurement board controller) to manage the readings, and the start and stop commands.

For clarity, here is an abstracted view of how a job is run from the point of view of a node:

1. Receive a job from the master node.
2. Setup the job environment (typically decompressing files and moving them to the correct directories).
3. Send a message to the measurement board controller to begin energy readings.
4. Run the job.
5. Send a message to the measurement board controller to stop energy readings.
6. Request the total energy reading from the measurement board control.
7. Send the output of the job and the energy reading back to the master node.

Once the job is complete, the master node is responsible for returning the energy readings and the output data to the user or process that requested it. This setup reduces the cost of evaluating many thousands of evaluations and can easily be expanded if needed. The relatively inexpensive components are an advantage compared to alternative approaches, allowing for more nodes than would otherwise be possible.

3.2.2 Applications Selected

The tool we used to generate the Langdon format required all software to be written in C/C++ with a license permitting its use for experimental purposes, and, as evaluation took place on a Raspberry Pi device running the Raspbian OS, the software had to be compilable within this environment.

Due to inevitable overheads associated with sending energy measurement start and stop commands over a network [110], we chose applications that had a non-trivial execution time, which we defined as being greater than 5 seconds. The larger the execution time, the smaller the overheads are as a percentage of total energy consumption.

We limited the selection further to applications that could be run via command-line, have test cases (or applications in which they can easily be generated), provide a deterministic output for any given input and, once execution has started, do not require further user interaction. In choosing these applications, we consulted relevant literature on energy optimisation and found the PARSEC benchmark suite had been utilised frequently [92, 154]. We therefore decided that applications from this suite should make up part of our selection. To avoid having applications entirely from a single source, we limited selection from the PARSEC benchmark suite to two applications then searched open-source repositories, such as GitHub and SourceForge, for the remainder. In addition to the criteria outlined above, we diversified the application domains in our corpus, searching until we found applications in each of the following application domains: file compression, video processing, database processing and image processing.

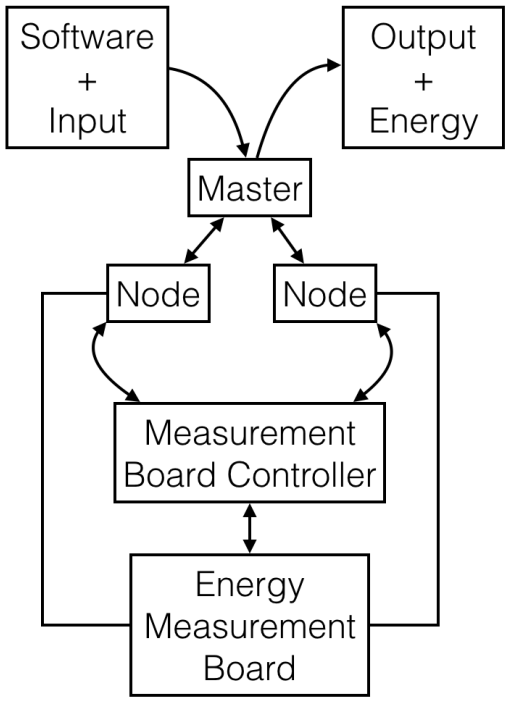


Figure 3.5: Diagram of the energy measurement cluster, showing two nodes measured by a single energy measurement board.

We selected from these domains as they are both important and popular. Furthermore, we believed these domains were likely to satisfy our aforementioned requirements, particularly in that they are all domains known to have non-trivial execution times for inputs that can be easily obtained or generated.

Table 3.5 shows the applications' domain, their size (in lines of code, LoC), and the number of lines tagged as modifiable in the Langdon format. In the following subsections we explain in more detail what these programs do, which parts we targeted for optimisation, and what input data we used.

Application	Domain	LoC	Modifiable LoC
7zip	Compression/decompression	136,828	2,524
Ferret	Image Search-Engine	13,260	5,032
Bodytrack	Body tracking	3,020	1,030
OMXPlayer	Media Player	14,164	5,184

Table 3.5: The Applications.

7zip

7zip Version 9.38.1 (Unix/Linux port) [1] is an open source file archiver with its own `7z` archive format. It consists of 136,828 lines of C/C++ code spread over 400 files. For the experiments outlined in this section, we concerned ourselves only with the core Lzma compression and decompression algorithms for optimisation. Excluding files associated with user I/O behaviour, we identified 6 files (`7zCrcOpt.c`, `LzFind.c`, `Lzma2Dec.c`, `Lzma2Enc.c`, `LzmaDec.c` and `LzmaEnc.c`) that accounted for over 99% of execution time when compressing and decompressing a 50MB text file. These files contain 6,258 lines of C code, 2,524 of which were tagged as modifiable when converted into the Langdon format.

40 test cases were used to evaluate 7zip. 10 audio files, 10 text files, 10 image files, and 10 large files. The latter included files and directories which ranged from 22.2MB to 64.4MB while the other three categories contained files with sizes ranging from 546KB to 12MB. Modifications made to 7zip were evaluated by measuring the summation of the energy required for the 7zip variant to compress then decompress each test case. For a test case to pass, two criteria must have been met. The test case (a file of directory) must have been compressed and that compressed file equal (in both size and composition) to that produced by the original, unmodified application. Secondly, the 7zip variant must have been able to decompress the compressed file back to its original state. We ran 7zip using `./7za a test.7za {test}` to compress and `./7za x test.7za -o ./output/` to decompress.

Ferret

Ferret is an image search engine. The program takes an image database and an image query as inputs then searches the database for images similar to the image query and returns the top candidates ranked by relevance (the number dependent on configuration). Ferret is part of Princeton's Parsec Benchmark Suite [31] and has previously been used as a candidate for genetic improvement at the machine-code level by Schutle et al. [154]. We used the most up-to-date version of Ferret at the time of carrying out our investigation; that contained within Parsec 3.0. Ferret is made up of 52 C/C++ files (excluding libraries) which contain 13,260 lines of code. When the Langdon format is applied 5,032 lines of code are deemed as modifiable. Due to Ferret's relatively small size we chose to optimise the entire application.

We used the 'simlarge', 'simmedium', and 'simsmall' test cases provided as part of the PARSEC Benchmark Suite to test the application. The 'simlarge' runs 256 image queries on a database of 34,973, the 'simmedium' runs 64 images queries on a database of 13,787 images, and 'simsmall' runs 16 image queries on a database of 3,544 images. Each of these queries will return a top 10 ranking. For a test to be considered as passed, it must have produced the same rankings, for each query, as the original, unmodified application. We ran Ferret using `./parsecmgmt -a run -p ferret -i {test}`, and measured the cumulative energy cost in running all three tests to determine a variant's fitness.

Bodytrack

Bodytrack is a computer vision application that tracks a human body through an image sequence. The application is capable, without markers or human involvement, to recognise a body's position within a video sequence from an array of cameras over a series of frames. It adds boxes to these images to produce a human-readable output. It is also part of Princeton's PARSEC Benchmark

Suite [31], version 3.0. Excluding libraries, Bodytrack consists of 23 C++ files that, in total, contain 3,020 lines of code. When the Langdon format is applied 1,030 lines of code are modifiable.

Bodytrack comes with three test sets which we used to evaluate solutions: 'simsmall', 'simmedium', and 'simlarge'. The 'simsmall' test set consists of 4 cameras, each of which take 1 frame of footage. 'simmedium' has 4 cameras and takes in 2 frames of footage. 'simlarge' has 4 cameras and takes in 4 frames of footage. The output for each is a series of points which can be plotted on the input frames to highlight the location of a body within it. For a test case to be passed, Bodytrack must have returned the same points as the original, unmodified application. We run Bodytrack using `./parsecmgmt -a run -p bodytrack -i {test}`, and measured the aggregate energy cost in running all three test cases as its fitness value.

OMXPlayer

OMXPlayer [8] is a video player operated via command-line interface. It takes a video file as input and outputs the necessary data to the HDMI port. Of particular interest to this investigation is that OMXPlayer has been specifically designed with the Raspberry Pi hardware in mind as it takes advantage of the Raspberry Pi's GPU. It thereby differs from the other candidates that exclusively interact with the traditional computer architecture.

OMXPlayer consists of 14,164 lines of code spread over 24 C++ files. This excludes the `FFmpeg` package which, though included in the source code, and necessary for execution, functions as a third-party library to the application. The number of lines tagged as modifiable using the Langdon format was 5,184.

The tests for OMXPlayer consisted of 10 MP4 video clips gathered from `https://archive.org`. The videos' average length was 14.7 seconds with a minimum of 13.0 seconds and maximum of 15.0 seconds. In order to evaluate modified versions of OMXPlayer, the application was modified to copy the data that would have sent through the HDMI interface to a text file; one HDMI packet per line. As this writing to file may have had some impact on energy consumption, all OMXPlayer variants were run twice. Once with the HDMI-to-textfile functionality and again without. The latter is when the energy measurement was taken; the former was used to record HDMI packet data. A test was said to have passed if the HDMI packets output were identical to that produced by the original, unmodified application. We ran OMXPlayer using `./omxplayer -p -o hdmi {test}`, and measured the energy consumption of playing all 10 video clips as the fitness value.

3.2.3 Experiment Setup

We ran the same GI algorithm as the one discussed in Section 3.1 though with a different fitness function. Instead of using the Intel Power Gadget to obtain energy estimates we utilised the cluster outlined in Section 3.2.1. We observed that energy readings given by the MAGEEC energy measurement boards varied though proportional increases and decreases remained constant (we investigate this in greater detail in Section 3.3). Therefore, for each application we report the proportional change in energy consumption as the solutions fitness. As our energy measurement cluster enabled many solution evaluations to be carried out at once (thus decreasing the 'cost' of evaluation in our GI algorithm), we ran all test cases relevant to that application for each solution. We therefore did not utilise the 'binning' technique used within the initial investigation. The 'champion' solution in our experiments was therefore the fittest solution found within the final generation. Unless mentioned here, all other aspects of the GI framework are as they were in the previous, initial study.

Application	Num Gen	Pop Size	Run #	Energy reduction	p-value
7zip	20	100	1	N/A	0.419
Bodytrack	20	100	1	N/A	0.461
Bodytrack	20	100	2	0.92%	3.39×10^{-4}
Bodytrack	20	100	3	1.20%	2.12×10^{-5}

Table 3.6: Summary of the experimental runs attempted.

3.2.4 Results and Discussion

As previously mentioned, this study did not produce variants that were significantly more energy efficient than the original, unmodified applications. As such, it was abandoned before all applications were run. However, here we present the results run and our interpretations of them.

Table 3.6 shows the runs attempted, highlighting the percentage energy reduction and the Wilcoxon rank sum test with continuity correct p-value. We ran 7zip for 20 generations with a population size of 100 and found that the champion solution (the best solution of the last generation) did not produce a statistically significant decrease in energy consumption when compared to the original, unmodified application (determined using the Wilcoxon rank sum test, against each application, original and modified, ran 100 times on the full test set with their energy consumption measured). Disappointed by this result we moved onto Bodytrack, again running for 20 generations with a population size of 100. As with 7zip, for the first run of Bodytrack we did not observe a statistically significant reduction in energy consumption.

We then decided to run Bodytrack two more times (with different PRNG seeds) to see if these disappointing results were simply bad luck or indicative of a wider problem. These two runs did produce statistically significant results ($p < 0.01$ in both cases) but only reduced energy consumption by around 1%; a result we did not believe to be that impressive or close to what we desired (typically, in energy optimisation of software systems, reductions of around 20% are feasible [39, 43, 154]).

These disappointing results made us pause and ask whether significant energy reduction using GI was, with this setup, possible. In the previous investigation (Section 3.1) the GI had, by proxy, achieved energy savings by tweaking program parameters and removing complex assert statements. Optimisations like this are unlikely to be universal. Programs may not have such influential parameters so easily accessable in the source code or contain complex, inconsequential statements that can simply be removed without influencing parameters. The question we then raised as this stage was *typically, what reduction in energy consumption is possible when using this approach to GI?*

Fascinated by this question we decided to terminate this research and conduct a larger, more rigorous investigation into the nature of the energy optimisation search space when using GI which we have documented in the following section.

3.3 Oracles And Synergy

Given research into optimisation of software's energy efficiency outlined in the previous two sections, it was determined that very little was known about the search space that was being navigated and for GI to progress in this field a better understanding of the code change search space is required.

Our first observation is that GI research, hitherto, has been dominated by three operators:

delete, *copy*, and *replace* which are applied at the source code line level [2] [108, 109, 145]. The *delete* operator deletes a line of code; *copy* copies a line of code to another location; and *replace* replaces a line of code with another. The challenge in GI research is designing search techniques to select a subset of all possible modifications that may then be applied to the target software to produce an optimal (or near optimal) solution. Until now there has been little effort put to analysing the search space these operators produce, and that must be subsequently traversed, when optimising software's energy consumption. This is unfortunate as GI practitioners have much to gain from understanding the characteristics of these search spaces.

The *delete*, *copy*, and *replace* operators generate an infinite search space, bounded by the number of *copy* operator applications. Even when restricted to a single operation, the search space remains large. For a program with N lines of code, every line can be deleted (N), copied into the program before existing lines (N^2), or replaced with any other line but itself ($N^2 - N$). In our study, the smallest application we investigate, Bodytrack, has 1,030 modifiable lines of code and, thus, over 2 million possible variants generated by the application of a single operator. GI pracitioners typically restrict this search by selecting a subset of the software system for modification. This subset is usually chosen by an expert with intimate knowledge of the system, or via profiling; selecting lines/files/components/etc. based on their likelihood of impacting the target non-functional property. In practice, even this restricted search space remains vast. The necessity for well-designed search techniques is clear though the information required to effectively design them is not presently available. The aim of the investigation documented in this section was to gain greater understanding of the search space and considerations researchers should take when optimising software's energy consumption using GI.

One such consideration is the oracle; a construct that can be used to verify a modified software's correctness [28], and typically, in GI research, takes the form of a well-designed test-suite. In this section, we focused on opportunities for energy improvement under two different test oracles — exact and approximate. An exact test oracle requires the original and improved programs to produce an identical output (as we used in Sections 3.1 and 3.2) while an approximate test oracle uses a more relaxed notion of whether the output from the improved program is acceptable. In this section we have used test suites to determine the correctness of an application and therefore we wish to emphasise that, when we say 'exact' or 'approximate', it is exact or approximate *modulo a test suite*. Each of these test oracles produce their own search space, both applicable to GI research and both worthy of study. Approximate test oracles permit trading quality attributes against energy consumption, which previous work on energy improvement (using GI and other techniques) has shown effective. For example, a mobile application can trade the aesthetics of a user-interface [116] while a graphics-based application can trade image quality [158]. In such cases, deviation from the precise output of the original may be tolerable if a decrease in energy consumption is observed.

For this study, we analysed the search space of the same four systems targeted in Section 3.2 — 7zip, Bodytrack, Ferret, and OMXPlayer. For each, we define, justify, and investigate approximate oracles that make domain-specific trade-offs between energy consumption and solution quality. We were interested in knowing at what frequency effective modifications exist in this search space, what impact they are capable of producing, and how this varies between exact and approximate test oracles.

Most previous energy optimisation work in software engineering has used *indirect* measures of energy consumption. Examples are tools which estimate energy by logging processor states [43], monitoring bytecode execution [47], or via simulation of hardware [179]. They interpolate energy

[2] Other GI work also modifies software at the binary and assembly levels [106, 154].

from correlated measurements. Indirect measurements are typically close to actual energy consumption, but their error is often unknown. Given that improvements reported hitherto are relatively modest (in the range of a few percent to a few tens of percent), it is important to quantify measurement error. To this end, we conducted our experiments on the energy measurement cluster, previously outlined in Section 3.2.1.

Our cluster enabled us to distribute software variants across different physical devices. As such we could take many more measurements than we would otherwise be able to, and can thereby quantify statistical error, like 'background noise', by reporting on the averages found over many runs. A key finding of ours is that individual devices exhibit systematic error. We found energy changes reported in Joules can vary considerably across different devices even when the statistical error within a single device is small. In future work, it is paramount that such systematic error is properly addressed. Within this investigation we found the proportional change in energy measurements are stable across all devices and therefore report results as proportional increases or decreases.

This setup enabled us to understand the properties of the energy search space by measuring the energy consumed when running software modified by the *delete*, *copy*, and *replace* operators. We could analyse both the local neighbourhood (a single modification) and beyond (multiple modifications), allowing us to give insight to GI practitioners.

If we were to find the local search space is flat (i.e. a single modification is incapable of, or rarely produces, a significant proportional change in energy consumption), then we could conclude that either the *delete*, *copy*, and *replace* operators are relatively ineffective or a highly explorative search technique is required to optimise software. Alternatively, if we were to find the local search space to be on a steady gradient, then we would argue the search-based algorithms should be based on exploitation (such as a hill-climbing algorithm) and, depending on the incline, we could state that GI researchers intuitions are correct — the *delete*, *copy*, and *replace* are effective.

The nature of the wider search space can be determined by combining modifications and noting their interaction. In our investigation, we observed instances when adding or removing a set of modifications produced a good solution but adding or removing a subset of those changes produced a much less effective solution. We refer to this as *synergy*, a specific form of interaction where the improvement of simultaneously applying multiple modifications exceeds the sum of applying each in isolation [29]. We also observed *antagonism*, another form of interaction that is the opposite of synergy. This occurs when the effectiveness of a solution worsens as modifications are combined in comparison to when they are applied individually. If antagonism is infrequent, then a greedy approach would be sufficient in combining modifications; simply sample modifications at random, evaluate them and, if they are found to be effective, add them to a list of good mutations to then be applied en-masse at the end of the process. In our investigation we found that antagonism occurd in 38.5% of all modification pairings — a frequency high enough to justify more advanced search techniques.

3.3.1 Motivating Example

The key to understanding the search space of energy-efficient software optimisations is to know at what frequency effective modifications occur, what impact they are capable of producing, and whether synergy or antagonism are common. We found these using both approximate and exact test oracles. This following provides a motivating example to explain these concepts.

In Figure 3.6, 'Mod_1' swaps a method that aggregates a list (at line 9) with one that samples. This increases the approximation of `getPropery`'s output but may achieve considerable

```
1  Property getProperty (List<Summary> summaries){
2    List<int> values;
3    for(Summary s in summaries){
4      values.add(s.getValue());
5    }
6
7    return new Property(
8      //Mod_1: aggregate → sample
9      new aggregate(values)
10   );
11 }
12
13 int sample (List<int> input){
14   input=sort(input); //Mod_2: Delete this line
15   return input.get(random(0, input.size())) * input.size();
16 }
```

Figure 3.6: An example of two software modifications.

energy savings because of sampling's relative efficiency. This is the type of modification that an approximate test oracle allows.

If we further assume the input to the method `sample` is sorted, then line 14, `input = sort (input);`, is not required. The software engineer responsible for this line may have included it to ensure robustness or due to a lack of knowledge about the contract that the `sample` method obeys. Regardless, guided by a sufficiently adequate test suite, GI can remove such redundancies when using an exact test oracle. In previous GI work by Petke et al. [145] and in the investigation outlined in Section 3.1, such optimisations were found when deleting complex assertions in MiniSAT's `Solver.c` class. The 'Mod_2' example is similar to this; an exact test oracle can find the modification, since it does not affect the software's output, only its target non-functional property.

It is tempting to pursue the modifications found by the exact oracle exclusively, as they produce benefits without cost. However, if we allow the quality of output to degrade (i.e. permit an approximate output), then this should increase the set of valid solutions in the search space and facilitate the search for even more energy-efficient solutions. Figure 3.6 demonstrates synergistic software modifications. 'Mod_1' decreases the number of times in which the more energy inefficient method `aggregate` executes by replacing it with `sample` while 'Mod_2' increases the efficiency of `sample`.

The equations below explain the basic mathematics of this synergistic interaction. The energy consumed by the program m_p equals the sum of m_g, the energy consumed by `getProperty`, multiplied by N_g, the number of runs, and m_s, the energy consumed by `sample`, multiplied by N_s, the number of samples. In our example, $N_g \geq 1$ and $N_s \geq 1$. Activity outside these methods is assumed to be constant and is represented by m_o and thus we have

$$m_p = m_s N_s + m_g N_g + m_o.$$

'Mod_1' changes `getProperty` to call `sample` instead of `aggregate`. The energy consumption of `getProperty` thereby includes the energy of a single iteration of `sample` plus the remainder of `getProperty` minus the call to `aggregate`, m_a. So when Mod_1 is applied,

$m_g \rightarrow m_g - m_a + m_s$ and we have

$$m_p = m_s N_s + (m_g - m_a + m_s) N_g + m_o$$

'Mod_2' decreases the energy consumption of `sample`, m_s. With 'Mod_1' present, energy is reduced in both $m_s N_s$, and in `getProperty`, formally $m_g N_g$ is now $(m_g - m_a + m_s) N_g$. Without 'Mod_1', 'Mod_2' only affects the energy consumed in `sample`, however, with 'Mod_1', 'Mod_2' may reduce energy consumption in both functions. In this investigation, we wish to understand how frequent these synergistic (or, the opposite — antagonistic) interactions occur within the search space.

3.3.2 Methodology

Here we discuss how we compared the effectiveness and energy efficiency of an original program and one of its variants, under both exact and approximate test oracles. We also introduce the system we used for classifying interactions between modifications.

Assessing Individual Modifications

As we noted in the start of this section, the search space of possible modifications is vast, too vast to analyse exhaustively. We therefore chose to uniformly sample it.

Formally, we define a modification as follows:

Definition 1 (Program Modification)**.** *A program modification is a pair* $\delta = (e, \vec{l})$ *where* $e \in \{$*copy*,*delete*,*replace*$\}$ *and* $\vec{l}$ *is a pair of program locations.*

Remark. We require locations $\vec{l}$ to be a pair, since *copy* and *replace* operations require two program locations.

We applied modifications chosen uniformly at random to lines tagged as modifiable in the Langdon format (let this number be n). We then added individual modifications that compile to what we refer to as the *Modification Set* until the cardinality of this set was $2n$.

Even when sampling, we did not wish to evaluate every variant against all available test cases. Variants that were inert, produced software that breaks hard-constraints or increased energy consumption were irrelevant to our investigation. Therefore, we filtered them out.

Algorithm 4 presents the filtering algorithm we used. The algorithm evaluates members of the modification set, E. First, it uniformly selects a test case t from a the set of test cases T. Then, the algorithm runs the original program P using test t with its energy measured, via function m, N times (100 in this case). The set of energy measurements M_P is then used to determine the 95% confidence interval lower bound, b_l. For each modification δ in the modification set E, we apply the modification to the program thereby creating a program variant P'. We then record the output of the program O'_p, and measure the energy consumption, J (Joules), of this program variant when producing this output. If the variant passes the test case (validated using the application's oracle) and its energy consumption is less than the 95% confidence interval (CI) of the mean lower bound, we add the modification to the *Candidate modification set*, E'. After this filtering step, the Candidate modification set contains those modifications for which we can say, with statistical confidence, that an improvement in energy efficiency has been observed while still passing the test case.

Algorithm 4 The Filtering Step.

Require: E, a set of modifications (Def. 1)
P, the target software
T, the set of test cases
N, the number of energy measurements
uniformSelection(T), uniformly selects $t \in T$
$CI(Mp)$, returns the 95% CI upper and lower bounds for a set of measurements Mp
applyMod(P, δ), returns modified program P' (program P modified by patch δ)
testOracle(O_P, t), returns true if the program's output, O_P, is correct for test t

1: $t \leftarrow$ *uniformSelection*(T)
2: $M_P \leftarrow \{\}$ # A set of energy measurements
3: **for** $1..N$ **do**
4: $\quad M_P \leftarrow M_P \cup \{m(P(t))\}$
5: **end for**
6: $[b_l, b_h] = CI(M_P)$ # We discard b_h
7: $E' \leftarrow \{\}$
8: **for** $\delta \in E$ **do**
9: $\quad P' \leftarrow$ *applyMod*(P, δ)
10: $\quad J \leftarrow m(O'p \leftarrow P'(t))$
11: $\quad$ **if** *testOracle*$(O'_P, t) \wedge J < b_l$ **then**
12: $\quad\quad E' \leftarrow E' \cup \{\delta\}$
13: $\quad$ **end if**
14: **end for**
15: **return** E'

It should be noted that 'passing' a test case in this instance does not necessarily mean producing the same output as the original, it may be approximated. For example, in the case of 7zip to pass a test case, the application must compress the test case in a manner that it may be decompressed to its original state though the compressed file generated by a program variant is permitted to differ from that produced by the original application. Section 3.3.4 precisely describes the criteria used for the exact and approximate test oracles. These criteria determine whether the software variants pass or fail for a given input.

Algorithm 5 The Evaluation Step.

Require: E', the set of modifications (Def. 1)
P, the target software
T, the set of test cases
N, the number of energy measurements
applyMod(P, δ), returns modified program P' (program P modified by patch δ)
testOracle(O_P, t), returns true if the program's output, O_P, is correct for test t
getApproxVal(O_P, O'_P), compares outputs of the original and modified programs then returns that Approximation Value

1: $D \leftarrow \{\}$ # Collection of modification data
2: **for** $t \in T$ **do**
3: $O_P \leftarrow P(t)$
4: **for** 1..N **do**
5: $J \leftarrow m(P(t))$
6: $D.addRecord(\perp, t, J, 0, \text{true})$
7: **end for**
8: **for** $\delta \in E'$ **do**
9: $P' \leftarrow applyMod(P, \delta)$
10: **for** 1..N **do**
11: $J \leftarrow m(O'p \leftarrow P'(t))$
12: $p \leftarrow testOracle(t, O'_P)$
13: $a \leftarrow getApproxVal(O_P, O'_P)$
14: $D.addRecord(\delta, t, J, a, p)$
15: **end for**
16: **end for**
17: **end for**
18: **return** D

Algorithm 5 presents the pseudocode that explains how we gathered data to evaluate the candidate modification set. In the algorithm, for each test case, t, the unmodified software is run N times (N was 30 in our investigation) with its energy, J, measured on each iteration via function m. Then, again for each test case, in line 9 each candidate modification δ is applied to the unmodified software to produce the modified variant, P'. The modified variant then processes the test case with its energy J measured and its output O'_P recorded for N iterations. We subsequently use this data to determine whether a modification produces a statistically significant reduction in energy consumption, using the Mann-Whitney U test (we declare statistical significance at $p < 0.05$).

At line 12 of the algorithm, we record whether the software variant has passed the test case, and, at line 13, we determine the *Approximation Value*, a, our unified approach to recording values

from both exact and approximate oracles. We also determine whether the software variant has passed the test case, p. The formula for both the approximation value and what passing a test case means for each application is defined in Section 3.3.4.

In all cases, an approximation value of zero denotes satisfaction of the exact oracle — that is, the output of the modified program correspond to the output of the original program (modulo the test cases). However, the results of the approximation value can be non-zero, with higher approximation values corresponding to greater degrees of approximation. The calculation of the approximation value is unique to the application domain and therefore approximation values from different applications cannot be directly compared. If a new application were to be introduced, the calculation of that application's approximation value would have to be created by an expert with domain knowledge. This common terminology serves to combine very different measures of approximation. We define the four domain-specific approximation criteria we used in this investigation in Section 3.3.4.

Classifying Interactions of Multiple Modifications

With this investigation, we wished to study how modifications interact. In obtaining the data to do so, we took two effective modifications (those known to pass all tests and reduce energy consumption) and measured their energy when applied to a piece of software both individually then when combined. The difficulty lies in interpreting the results we obtained. In order to do so, we labeled a modification pair in accordance to its place within an *Interaction Spectrum*, outlined in the following definitions.

Definition 2 (Patch)**.** *A patch Δ is a non-empty sequence of modifications δ (Def. 1).*

Remark. In the case where a location within the software is modified, and that location is subsequently used by another modification later in the sequence, the new value of that location is used (i.e. the value of that location after the preceding modifications have been applied). For example, if line X is deleted via the *delete* operation, and then line Y is replaced with the value of line X via the *replace* operation, it is a deleted (i.e. blank) line that line Y is replaced with.

Definition 3 (Interaction Spectrum)**.** *Let $r(P_\Delta)$ denote the reduction in energy measurement when patch Δ is applied to program P. Formally, $r(P_\Delta) = m(P) - m(P_\Delta)$, where m is our energy measurement function. Two patches Δ_1 and Δ_2 interact when the measure of their joint and independent application differs. Formally, $r(P_{\Delta_1\Delta_2}) \neq r(P_{\Delta_1}) + r(P_{\Delta_2})$. Figure 3.7 shows this* interaction spectrum *partitioned into three categories:*

$$\textit{Synergy} : r(P_{\Delta_1\Delta_2}) > r(P_{\Delta_1}) + r(P_{\Delta_2})$$

$$\begin{aligned}\textit{Weak Antagonism} : r(P_{\Delta_1\Delta_2}) &< r(P_{\Delta_1}) + r(P_{\Delta_2}) \\ \wedge\ r(P_{\Delta_1\Delta_2}) &> \max(r(P_{\Delta_1}), r(P_{\Delta_2}))\end{aligned}$$

$$\begin{aligned}\textit{Antagonism} : r(P_{\Delta_1\Delta_2}) &< r(P_{\Delta_1}) + r(P_{\Delta_2}) \\ \wedge\ r(P_{\Delta_1\Delta_2}) &< \max(r(P_{\Delta_1}), r(P_{\Delta_2}))\end{aligned}$$

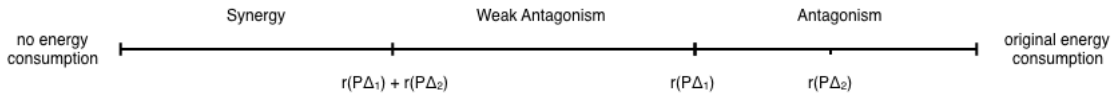


Figure 3.7: The interaction spectrum (Definition 3).

Remark. We classify an interaction as synergistic when the joint application of two patches has a greater effect than assuming no interaction. When weak antagonism occurs, we accept that the patches interact in a way as to dilute their effects but not in a manner that precludes their joint application. When interactions exhibit synergy or weak antagonism, both patches should be applied because the reduction in energy from their joint application exceeds either applied alone; if the patches exhibit strong antagonism, the most effective patch should be chosen and the other discarded.

3.3.3 Research Questions

Any attempt to improve energy consumption, search-based or otherwise, relies on the ability to reliably measure energy. Our first question therefore investigated the degree to which our energy measurements were sufficiently reliable to assess energy improvement:

RQ1, Measurement: What variance occurs when measuring energy consumption?

As with all forms of real-world measurements, energy measurements are vulnerable to a number of different sources of variation. With this in mind, we wished to establish the degree of variance to expect, both within a single energy measurement device and across multiple devices. Even on a single device, the amount of energy consumed may vary when, on different occasions, exactly the same software system is executed with exactly the same test suite; we wished to understand the magnitude of this variance. If the variance was found to be high, then we would have no foundation upon which to make reliable measurements. We argue that *any* experimental work on energy assessment or improvement should, as a preliminary step, report results for such variance, in order to exclude a serious potential threat to validity of the scientific findings. This motivated our first research question:

RQ1a: What is the variance when measuring using a single energy measurement device?

To answer this question, we chose a node within our cluster as a test target. Then for each application, we uniformly selected a test case and executed the application 30 times on the target node, recording the energy consumed during each iteration. We used this data to measure the variance within devices. This variance informed us of the statistical error in the measurements we obtained.

Even if the variance within the devices was found to be small, there may still be variance between different devices. Several studies of energy assessment and improvement have reported results based on only a single device [22, 115, 121]. This leaves open another potential threat to the validity of the findings, which occurs if different instances of the same device type give highly different readings for the same software system and test suite. While RQ1a informs us of the statistical error, we may miss detecting a form of systematic error where different devices give different measurements for the same process. This motivated RQ1b:

RQ1b: What is the variance in direct energy measurements across multiple devices?

To answer RQ1b, we uniformly selected an application and a corresponding test case. We then

ran that application and test case pair on *all* the devices in the cluster, 100 times, measuring the energy consumption each time. We used boxplots (one boxplot per device) to determine if there was variance in energy measurements across the devices.

Our goal in answering RQ1b was to find whether there is systematic error across different devices, though, when present, this error *can* be tolerated if it is consistent as we are only interested in the *proportional* differences in energy consumption when assessing energy improvement, not the *absolute* measure of energy consumed. This motivated RQ1c:

RQ1c: What is the variance in proportional energy changes across multiple devices?

To answer RQ1c, we uniformly selected an application and all of its test cases. We then ran the application with all of its test cases on every device in the cluster. For each device, starting with the test case which consumed the smallest amount of energy, we recorded the proportional increase in energy consumption between it and the next smallest test case. We then created a boxplot for each of these proportional increases across all devices to show whether these proportional figures were reliable across our cluster. Once we determined the suitability of our energy measurement cluster, we could begin evaluating our applications and the variants produced by applying modifications to them.

When assessing whether a modified program was acceptable or not, we needed to determine whether the behaviour of the improved program was acceptable with respect to the behaviour of the original. In software testing, more generally, this is an instance of the oracle problem which, though significant, is largely unsolved [28]. However, one of the advantages of genetic improvement is that the original version of the program can act as an oracle, against which improved versions are compared [86, 178]. For a given candidate program variant, we checked the variants behaviour against that of the original to see if deviation had occurred. This raises the fundamental question of how much deviation from the behaviour of the original can be tolerated.

For some application scenarios, no deviation can be tolerated, but, in many other scenarios, exact replication of behaviour is unnecessary. Previous work on genetic improvement has shown that genetically modified programs may improve not only targeted non-functional properties of interest, but also the functionality of the original program [108]. In such situations, the original program's behaviour only acts as a guide to the desired behaviour of the genetically improved program.

Furthermore, even when functional improvement is not possible, the genetically modified program may need only *approximate* the behaviour of the original, sacrificing some degree of result quality for improvements in non-functional characteristics. Often minor quality degradation is imperceptible or acceptable to the end user, making such trade-offs highly desirable. Much of the work on energy improvement falls into this category [92, 116, 128].

This motivated RQ2, which investigated using approximate test oracles, allowing us to trade solution quality against energy improvement:

RQ2, Improvement: What additional energy improvement can be achieved when using approximate test oracles in place of exact test oracles?

In answering RQ2, we investigated the degree to which energy efficiency can be improved by sacrificing solution quality, guided by a domain-specific approximate test oracle in each case. We also investigated the effect of approximate test oracles on the frequency and impact with which the different genetic operators affect energy consumed and the trade-off between energy consumption and solution quality.

Finally, we considered the way in which different genetic improvement modifications to the original program combine to improve energy efficiency. The motivation for this research question

Application	Domain	LoC	Modifiable LoC	No. of Modifications
7zip	Compression/decompression	136,828	2,524	5,000
Ferret	Image Search-Engine	13,260	5,032	11,000
Bodytrack	Body tracking	3,020	1,030	2,000
OMXPlayer	Media Player	14,164	5,184	10,000
Total				28,000

Table 3.7: Number of modifications investigated for each application studied.

derived from the way in which search-techniques typically combine lower-level building blocks of partially fit solutions in order to arrive at fitter combined solutions [57, 88, 126]. In RQ3, we therefore studied the interactions between individual modifications, reporting the frequency of different kinds of synergistic effects:

RQ3, Synergy: How frequently do synergistic and antagonistic effects occur when combining known effective modifications?

To answer RQ3, we performed a pairwise investigation of the modifications found to reduce energy consumption. We took 15% of all possible pairings from the set of effective modifications found in answering RQ2 (those found when using an approximate test oracle). We evaluated each and reported the frequency of synergy and antagonism observed in accordance to the interaction spectrum outlined in Section 3.3.2.

3.3.4 Test Subjects and Their Oracles

In order to answer these Research Questions, we chose to optimise Bodyrack, OMXPlayer, Ferret, and 7zip, the same four applications we detailed in Section 3.2. The applications, and the manner in which the Langdon format has been applied, is identical to what was described and used in Section 3.2. The main additions necessary for this investigation, were the creation of approximate oracles for each application, while also determining a separate 'passing' criteria for the testcases provided.

Table 3.7 (an expanded version of Table 3.5 from Section 3.2) shows the applications' domain, their lines of code (LOC), the number of lines modifiable in the Langdon format, and the number of modifications we generate and study. In the following subsections, we document how the approximate oracle works for each application, and how our 'passing' criteria is determined across both oracles.

7zip

For our experiments using 7zip, we generated a modification set of size 5,000. 7zip was evaluated by measuring the total energy required to compress and then decompress a test case. For a test to pass, the test must have been compressed and then decompressed to its original state. Unlike the investigation in Section 3.2, the state of the compressed file is irrelevant (as long as it could be properly decompressed). The approximation value was calculated using the compression ratio. Equation 3.1 shows how this approximation value was calculated. To calculate the approximation, the original program P compressed a test case, t, with the size of the compressed test case recorded (we overload $|\cdot|$ to denote file size). We did the same with the modified program P'. The approximation was the size of the compressed file produced by the modified program divided by the

compressed file produced by the original. This ratio had one subtracted so that a value of zero was returned when there is no change in compression rates. A higher approximation value indicated worse compression while a lower approximation value indicated better compression in the modified software.

$$\frac{|P'(t)|}{|P(t)|} - 1 \tag{3.1}$$

Ferret

We generated a modification set consisting of 11,000 modifications. Ferret was evaluated by measuring the total energy required to run one its test cases ('simlarge', 'simmedium', and 'simsmall'). For a test case to be considered a 'pass', a non-empty ranking for each query must have been returned. The calculation of the approximation value for these rankings is shown in Equations 3.3 and 3.4.

To calculate the approximation value for a given Ferret output, the output rankings produced by the original software P were compared against that produced by the modified software P'.

Then, for each query ($q \in Q$), the ranks were compared using the Kendall's τ ranking statistic. For equal rankings, Kendall's τ returns one and tends to negative one for more unequal rankings produced. Our approximation value rules required zero to be returned when no approximation had taken place and tend higher for more approximate solutions. Equation 3.3 therefore manipulated the Kendall's τ statistic to conform to this by incorporating a stretch factor s which we set to 5,000. This resulted in K, our modified ranking statistic, ranging from 0 for equal rankings to 10,000 for completely unequal rankings. The interval arithmetic [90] for K is shown in Equation 3.2. The approximation value was the average over all queries. If an image was ranked for the original output but not the modified output then it was added to the end of the modified ranking.

$$\begin{aligned} \tau(L_1, L_2) &= [-1, 1] && \#\textit{Range of } \tau \\ [0, 2] &= [-1, 1] + [1, 1] && \#\textit{Shift interval} \\ [0, 2s] &= [0, 2] * [s, s] && \#\textit{Stretch by } s \in \mathbb{R}^+ \end{aligned} \tag{3.2}$$

$$\begin{aligned} &2s - s(\tau(L_1, L_2) + 1) \\ &= [2s - 2s] - [0, 2s] && \#\textit{Complement by } 2s \end{aligned}$$

$$K = 2s - s(\tau(L_1, L_2) + 1) \tag{3.3}$$

$$\frac{1}{|Q|} \sum_{q}^{Q} K(P(q), P'(q)) \tag{3.4}$$

Bodytrack

A modification set of 2,000 elements was created to investigate Bodytrack. Like Ferret, Bodytrack has three test cases: 'simlarge', 'simmedium', and 'simsmall'. For a test to have been considered as 'passed', Bodytrack must have returned a wireframe model for each frame within the video sequence.

Algorithm 6 shows how Bodytrack's approximation values when calculated. It averaged the differences between the points produced by the original software, l_P, and the points produced by the modified software l'_P (both contain $1..N$ points) for any given input. The difference between two points was the sum of the difference in the x component plus the difference in the y component. An approximation value of zero meant the output was identical to the original and tended higher as the results became more approximate.

Algorithm 6 Bodytrack's Approximation Calculation.

1: $l_P \leftarrow P(t)$
2: $l'_P \leftarrow P'(t)$
3: assert(l_P.size() == l'_P.size())
4: $N \leftarrow l_P$.size()
5: **while** $l_P \neq \{\}$ **do**
6: $\quad (x, y) = l_P$.dequeue()
7: $\quad (x', y') = l'_P$.dequeue()
8: $\quad s \leftarrow s + |x - x'| + |y - y'|$
9: **end while**
10: **return** $\frac{s}{N}$

OMXPlayer

For OMXPlayer, a modification set of 10,000 modifications was generated. The test suite for OMXPlayer was a set of video sequences. For a test to be considered 'passed', OMXPlayer must have output HDMI packets.

As a reminder, each OMXPlayer evaluation involved running the application twice: once with a version of OMXPlayer that processed the video in a traditional manner (this is when energy was measured), and again with a version of OMXPlayer with the video output copied to a text file for comparison, one line per HDMI packet.

The approximation value for any given test case is shown in equation 3.5. It is calculated by taking the number of lines returned by a POSIX `diff` on the output generated by the original software (i.e. a list of raw HDMI packets), L_P, in comparison with the output of the modified software, L'_P, for a given test case. This was then divided by the total number of lines the original output. Therefore, zero was returned when the outputs were identical and tended higher the more approximate the output became. We used this as a proxy for video quality.

$$\frac{|\texttt{diff}(L_P, L'_P)|}{|L_P|} \tag{3.5}$$

3.3.5 Results

We measured energy consumption for both the original and all the 28,000 software variants (see Table 3.7) of the four test subjects presented in Section 3.3.4, using the methodology outlined in Section 3.3.2. Having thereby identified a set of effective (i.e., energy reducing) modifications, we uniformly sampled 15% of all possible pairwise combinations. We applied these, in turn, to the four applications under test and measured the energy consumption to check for synergistic or

antagonistic effects. We summarise our results and answer the research questions posed in Section 3.3.3 as follows.

RQ1: Measurement

The first research question is concerned with the reliability of energy measurements within our Raspberry Pi cluster. In particular, we ask: *what variance occurs when measuring energy consumption?*

RQ1a asked '*What is the variance when measuring using a single energy measurement device?*'. To answer this, we plotted the variance in energy measurement, within a single device, across all applications studied. This resulted in the boxplots found in Figure 3.8. The mean for 7zip is 48.45J with a standard deviation of 0.31, for Bodytrack, 100.19J with a standard deviation of 1.17, for Ferret, 363.25J with a standard deviation of 1.26, and for OMXPlayer, 57.17J with a standard deviation of 0.18. We therefore concluded that the precision of the energy measurement setup is sufficiently high for the needs of our investigation.

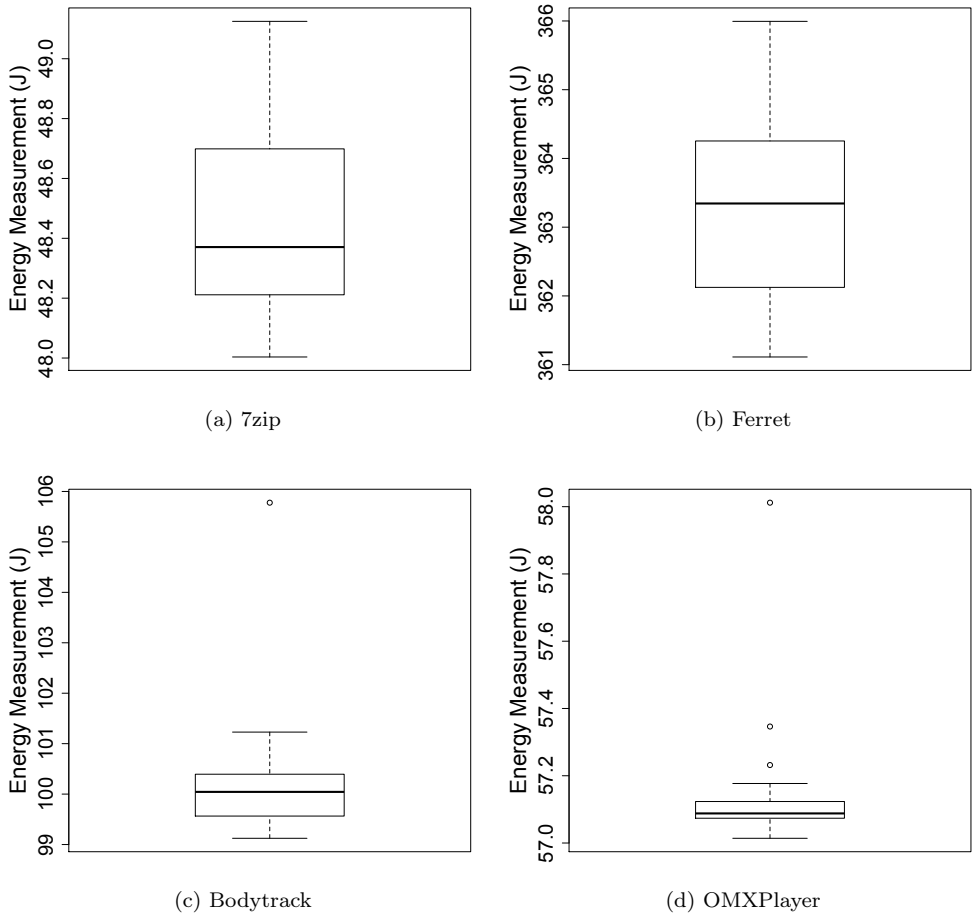
RQ1b asked '*What is the variance in direct energy measurements across multiple devices?*'. To answer this we measured the energy consumption of each node running 7zip (chosen uniformly at random from the four applications studied) on a single, uniformly chosen test case (in this case, 'The Complete Works of William Shakespeare', a 5.6MB file). The boxplots in Figure 3.9 show the distribution of energy readings for each Raspberry Pi device. As can be seen, the measurements varied noticeably between devices but are consistent within each device. In answering this research question, we also observed that restarting a node in the cluster resulted in different readings compared to those given before its restart. This did not interfere with our experiments as readings between restarts were consistent. A survey of the relevant literature unearthed analysis by Kalibera et al. [98] which describe this phenomenon as a little known, but none-the-less near-universal, problem when taking measurements of modern computer systems. They observed the significant differences that can occur when rebooting is primarily due to non-deterministic properties in modern operating systems, particularly memory management which can have a knock-on effect on cache hit-rates. In line with our findings, the measurements are consistent between restarts. In order to avoid this reboot effect interfering with our experiments, we avoided restarting nodes during experimental runs.

Therefore, we concluded that this approach to measuring energy consumption produces results that *lack accuracy but have good precision* [162]. As Figure 3.9 shows, it is impossible for each reading to be accurate, but within each device, the readings are precise.

This inaccuracy, however, is only relevant if we want to obtain energy reductions or increases in Joules which, in this investigation, we did not. In our investigations, we wished to obtain the proportional change between software variants and, therefore, it is important to show that the proportional change observed in one device is consistent across all devices.

It is for this reason RQ1c asked '*What is the variance in proportional energy changes across multiple devices?*'. To answer this, we ran Bodytrack (uniformly chosen from all applications studied) with each of its three test cases ('simsmall', 'simmedium', 'simlarge') on all devices in the cluster and recorded the proportional increases in energy consumption between simsmall and simmedium, and simmedium and simlarge for each device within the cluster.

The boxplots in Figure 3.10 show these proportional increases between the three Bodytrack test cases. The difference between the simsmall and simmedium averages 56.64% with a standard deviation of 1.03, and the difference between simmedium and simlarge averages to 38.46% with a



(a) 7zip

(b) Ferret

(c) Bodytrack

(d) OMXPlayer

Figure 3.8: The variance in measurements that occurred when running each application (unmodified).

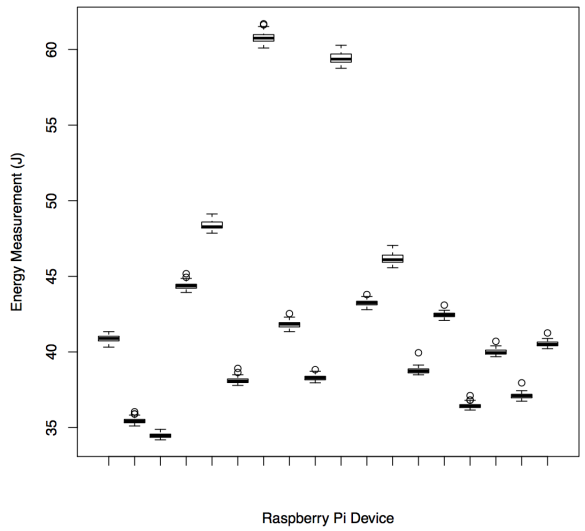


Figure 3.9: The variance in measuring the same program with the same input across different devices.

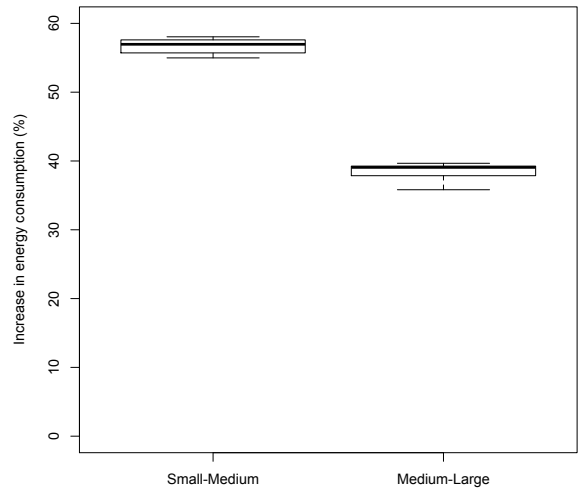


Figure 3.10: The variance in proportional energy change across different devices.

Application	+ve Mods	%age Mods	Max	Average
7zip	0	0.00%	N/A	N/A
Ferret	0	0.00%	N/A	N/A
Bodytrack	6	0.30%	2.69%	1.54%
OMXPlayer	4	0.04%	2.32%	1.51%
Average	2.5	0.09%	1.25%	0.76%

Table 3.8: Each application with the number and percentage of modifications that reduced energy consumption according to an exact test oracle.

standard deviation of 1.01. Given this small standard deviation, we believe that a proportional increase or decrease between two measurements within the same devices is consistent across all devices.

RQ2: Improvement

We were concerned with the trade-off between energy consumption and solution quality produced by modified software, which we obtained using the *delete*, *copy*, and *replace* operators. Therefore, we asked: *what additional energy improvement can be achieved when using approximate test oracles in place of exact test oracles?*

In order to obtain a baseline measurement, we first investigated the question: *what is the frequency and impact of energy-efficient modifications in the local neighbourhood when using exact test oracles?* To answer this, we extracted the data generated from the experimental procedure outlined in Section 3.3.2. We only considered a modification to be successful when it, on average, reduced energy consumption across 30 runs with this effect observed to be statistically significant ($p < 0.05$ according to the Mann-Whitney U test) for each test case. As we used exact test oracles in answering this research question, we excluded any modifications that had an approximation value not equal to zero.

Table 3.8 shows the results obtained to determine the frequency and magnitude of effective modifications in the local search space (i.e., that defined by the *delete*, *copy* and *replace* operators), assessed using exact test oracles. The most striking finding is the frequency of modifications that reduce energy consumption (i.e. '+ve mods'); averaging only 0.09% across all cases. The impact of these is an average decrease of 1.25% across all applications with a maximum of 2.69%. This is a significant finding as it indicates only small improvements can be found in the one-step local neighbourhood when using an exact test oracle.

Table 3.9 reports results obtained when approximate outputs were permitted. We analysed the same dataset but allowed modifications with a non-zero approximation value. As discussed in Section 3.3.4, a higher approximation value for 7zip means less compression; for Ferret, a greater inaccuracy in the search engine result rankings (using the original ranking as the baseline); for Bodytrack, a larger error in the plotting of the body's location within a series of images; and for OMXPlayer, a greater proportion of incorrect, or misplaced, HDMI packets. As our approximate oracles permit a significant degradation in output quality, it should be noted that the solutions which make up the averages in Table 3.9 may not be applicable to many real-world environments. As an example, a valid approximation for 7zip could be an alteration which results in it producing compressed files of equal or greater size than what was input. Evidently such a result would not be of use in any conceivable domain. What we wished to demonstrate here is that approximation

Application	+ve Mods	%age Mods	Max	Average
7zip	8	0.16%	48.24%	13.16%
Ferret	157	1.43%	79.88%	51.13%
Bodytrack	72	3.60%	33.69%	8.17%
OMXPlayer	24	0.24%	95.60%	63.15%
Average	65.25	1.36%	64.35%	33.90%

Table 3.9: Each application with the number and percentage of modifications that reduced energy consumption according to an approximate test oracle.

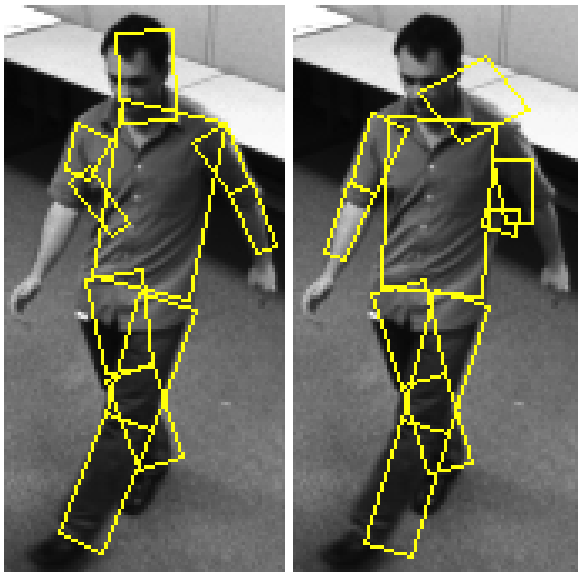


Figure 3.11: An output from the the original Bodytrack application (left), and and an approximated variant (right).

is an avenue for more optimisations; that the more a GI practitioner permits approximation, the more energy saving solutions can be found.

We found that approximation increased the frequency of effective modifications in the local search space to 1.36%; a 15-fold increase compared to those found when using the exact test oracle. This increase in frequency was mirrored in the increase in impact the average modification was capable of producing. While the average effective modification energy consumption reduction when using the exact test oracles was 1.25%, an average reduction of 33.90% was achieved when using an approximate test oracle.

As previously mentioned, these statistics assume any level of approximation is acceptable. Therefore, Table 3.10 provides Pareto frontiers for each of the applications investigated which show the energy reduction vs. output quality trade-off. In each, we consider all solutions which passed their respective tests, regardless of approximation. The approximation value and passing criteria in each case was calculated using the formulæ presented in Section 3.3.4. For each Pareto frontier, we describe what each Pareto optimal solution produces when run. We also sampled a

Energy Reduction	Approximation Value
5.08%	3.93×10^{-4}
12.29%	0.072
13.17%	0.102
48.30%	0.741

(a) 7zip

Energy Reduction	Approximation Value
43.19%	0.154
60.79%	39.873
75.53%	78.800
75.55%	1550.200
76.21%	2669.710
79.88%	6221.220

(b) Ferret

Energy Reduction	Approximation Value
2.69%	0.000
19.26%	0.131
27.97%	0.170
29.13%	0.192
33.69%	0.452

(c) Bodytrack

Energy Reduction	Approximation Value
2.32%	0.000
78.45%	0.003
92.70%	0.637
95.53%	1.002
95.60%	1.043

(d) OMXPlayer

Table 3.10: Pareto frontiers for the four subjects investigated.

single approximate solution for each application's Pareto frontier and attempt to explain why the modification reduced both energy consumption and output quality.

In the case of 7zip, the Pareto frontier contained 4 solutions, ranging from a 5.08% energy reduction with an approximation value of 3.93×10^{-4} to a 48.30% reduction with an approximation value 0.741. The latter translates to the compressed file generated by the modified software being, on average, 74% larger than if compressed using the unmodified version of 7zip. This variant of 7zip still performed non-lossy compression but less effectively. We analysed this solution in greater detail and found the modification deleted a line in `LzmaEnc.c` which initialised a variable declaring dictionary size (an unsigned 32 bit integer) to zero. There is then a method that follows which sets the dictionary size variable to 8MB *if* the dictionary size variable was initialised to zero. Otherwise, the dictionary size variable is kept at its specified non-zero value. As uninitialised integers produce undefined behaviour in C, the value of the dictionary size varied between runs though, from our observations, this was always significantly below the default 8MB figure. The highest we observed was 4KB. A lower dictionary size inevitably leads to less compression and a shorter execution and, therefore, less energy consumed overall.

For Ferret, there were 6 Pareto optimal solutions. This ranged from a 43.19% reduction in energy, for an approximation value of 0.154, to a 79.88% reduction in energy, for an approximation value of 6221.220. In the case of the former, the approximation value translates to a Kendall's τ of 0.73. The next solution on the Pareto frontier achieved a 60.79% energy reduction with an approximation value of 39.873. This approximation value translates to a Kendall's τ of -0.95, a value close to the Kendall's τ 'worst case' of -1. Therefore, five of Ferret's six Pareto optimal solutions produced solutions close to random (i.e. with very high inaccuracy). We sampled the solution with a 43.19% energy reduction and an approximation value of 0.154. This modification applied the *delete* operator to the condition of a `FOR` loop within `emd.c` (i.e. turned it to false). This turned off an energy intensive branch within Ferret's EMD (Earth Mover's Distance) algorithm. The EMD

algorithm computes the distances of colour histograms: Ferret uses it a metric to determine how similar two images are. With this modification, this distance is more approximate and therefore leads to a different ranking of images output for any given target.

Bodytrack had 5 Pareto optimal solutions. These ranged from a 2.69% reduction where there was no approximation to a reduction of 33.69% with an approximation value of 0.452. Figure 3.11 shows the most energy-efficient solution's output compared to that produced by the original, unmodified software. We sampled this solution to gain a greater insight as to how it achieved its approximation. Bodytrack functions by iteratively generating models (configurations of the wire-frame body, like that shown in Figure 3.11), assigning them weights proportional to their likeness of the body within the image. A subset of models are selected proportional to these weights and these models are 'mutated' by incrementing their parameters by random amounts as determined by a Gaussian distribution. This process is repeated until the maximum number of iterations is met or until the search converges on a model that has a sufficient likeness to the body within the image. Bodytrack determines the likeness of a model to the input image via an error function. The modification responsible for the 33.69% reduction in energy consumption deleted a line in Bodytrack's `ImageMeasurements.cpp` file which interferes with how this error is calculated, effectively lowering it. This allowed Bodytrack to converge on a more approximate model earlier, reducing execution time and, by extension, energy consumption.

Finally, the OMXPlayer Pareto frontier contained 5 Pareto optimal solutions. We visually inspected the video output when these modifications were applied. We found the three most approximate solutions did not produce video output which was viewable (a black screen with no audio). The next most approximate solution achieved a 78.45% reduction with an approximation value of 0.003. This approximation value means, on average, 0.15% HDMI packets differed from the output of the original application. We found that this solution was viewable but played video files at increased speed with distorted audio. The solution with no approximation and a 2.32% reduction in energy consumption, when visually inspected, was identical to the original as expected. We sampled the instance in which a 78.45% reduction in energy consumption was observed (with an approximation value of 0.003). We found that this was due to a *replace* operation in OMXPlayer's `OMXPlayerAudio` class; modifying an `IF` statement's condition, effectively turning its condition to true. This resulted in declaring audio to be 'passed through'. Pass-through is an option available in media players to turn off audio decoding and, instead, output encoded audio. This is desirable when the user wishes to offload decoding to more advanced hardware, such as home stereo setups. This explains why the audio was so distorted in our inspection of this modification. It also partway explains the energy reduction as this modification removes costly audio decoding. The reason for the video speed-up is less clear but appears to be due to how OMXPlayer processes video alongside audio. Each are allocated their own thread and, to ensure these threads remain in-sync, the video is sped up or slowed down to keep pace with the audio. Normally, these corrections would be unnoticeable to the user, speeding up or slowing down video stream by a small amount for a short period of time. As this modification results in the audio being passed through instead of processed, the audio thread runs considerably faster and thus the video speed is increased in a futile attempt to remain in-sync. If the user specifies audio pass-through when running the program via the command-line interface, checks are done to ensure the audio is passed-through to external decoders. If these checks fail, the user's decision is overridden. This modification bypasses these checks. We believe the increased video speed explains most of the energy reduction as it results in the total execution time of the video player decreasing.

Using the data gathered in this investigation, we were able to determine how frequently each of

	delete	*copy*	*replace*
7zip	5	0	3
Ferret	81	7	69
Bodytrack	44	1	27
OMXPlayer	8	3	13
Percentage	52.9%	4.2%	42.9%

Table 3.11: Number of effective modifications using the *delete*, *copy* and *replace* search operators across all applications.

	delete	*copy*	*replace*
7zip	16.60% (12.29%)	0.00% (0.00%)	7.42% (5.08%)
Ferret	56.45% (73.64%)	64.74% (75.17%)	43.50% (37.23%)
Bodytrack	8.27% (4.70%)	0.16% (0.16%)	8.31% (7.09%)
OMXPlayer	57.01% (71.09%)	71.86% (78.40%)	64.92% (78.44%)
Average	34.58% (40.43%)	46.59% (38.43%)	31.04% (31.96%)

Table 3.12: The mean and median (in parenthesis) impact of effective *delete*, *copy*, and *replace* modifications, in terms of % of energy reduction, across all applications.

the search operators occur in energy-saving modifications. Forest et al. [68] and Le Goues et al. [113] evaluated the *delete*, *copy*, and *replace* operators in the context of automated software repair. They found that *delete* is the most effective at 'fixing' bugs (Qi et al. [148] have since shown that many of these 'fixes' reduced symptoms rather than repaired bugs, however, this form of pseudo-repair may be sufficient in some circumstances), followed by *replace* with *copy* being considerably less successful. We found this trend largely holds when applied to genetic improvement for energy consumption. Table 3.11 shows the frequency of effective modifications (i.e., all modifications that reduced energy and conformed to the passing criteria outlined in Section 3.3.4), for each application studied, broken down by operator type. *delete*, followed closely by *replace*, are most likely to produce an effective solution. Table 3.12 shows the average and median energy reduction per operator type. This table shows that when copy is effective (albeit rarely as shown in Table 3.11) it can have the biggest impact, followed by *delete*, then *replace*.

In viewing these results, we were motivated to discover how many of the *replace* operations were, in effect, *delete* operations; *replace* operations in which the same effect could be obtained via a single *delete* operation. We uniformly sampled 20% of the effective *replace* modifications (those that reduced energy consumption and passed the tests, with approximation permitted) from each application and manually inspected them[3]. Our manual inspection process involved two software engineering researchers[4] examining each *replace* operation independently and classifying them as either 'delete-by-proxy' when it was decided the same effect was possible via the application of a single *delete* operation, or as 'genuine replace' when this was not possible. We also reserved a special 'unknown' classification if there was insufficient evidence from manual inspection of the source code to make a decision. Once both manual inspectors completed their classifications, the inspectors met and compared their decisions. Where there were conflicts in a modification's classification, the

[3]We sampled a minimum of one, for applications with less than five effective *replace* modifications.
[4]The manual inspection process was carried out by myself and Dr. Justyna Petke (my supervisor).

	Delete-by-proxy	Genuine Replace	Unknown
7zip	0	0	1
Ferret	7	4	1
Bodytrack	4	2	0
OMXPlayer	0	2	1
Total	13 (50%)	10 (38%)	3 (12%)

Table 3.13: A 20% sample of all effective *replace* operations manually classified as either effective *delete* operations, legitimate *replace* operations or unknown effect.

Application	synergy	weak antagonism	antagonism
7zip	0.9%	60.4%	38.7%
Ferret	9.2%	48.8%	42.0%
Bodytrack	35.3%	40.1%	24.6%
OMXPlayer	2.6%	48.7%	48.7%
Average	12.0%	49.5%	38.5%

Table 3.14: The percentage of effective modification pairings within the Interaction Spectrum (Def.3).

inspectors discussed their respective reasoning with the goal of coming to an agreement. In this case, the two inspectors reached agreement on all the modifications sampled.

We record the findings of this study in Table 3.13. We found half of all *replace* operations could have occurred through a single *delete* operation, though a substantial minority, 38%, were genuinely *replace* operations which could not have been recreated using a single *delete*. Referring back to Table 3.11, which shows 52.9% of all effective modifications were *delete* with 42.9% being *replace*, we can add more weight to the argument that *delete* is most effective as half of all *replace* operations are effective *deletions*.

RQ3: Synergy and Antagonism

This research question asked *How frequently do synergistic and antagonistic effects occur when combining known effective modifications?* We answered this question by selecting a subset (random uniform selection at 15%) of all available pairs of effective modifications (approximation permitted), evaluating them, then classifying them according to our interaction spectrum, as defined in Section 3.3.2. We selected a subset due to the cost of evaluation that would have occurred otherwise — analysing all effective modification pairs would have resulted in N^2 evaluations. Evaluating a uniformly selected 15% subset produced data that was sufficient to answer our research questions.

Table 3.14 shows the distribution of the interaction classifications. As can be seen, at 49.5% of all pairings, weak antagonism is the most common classification. 12.0% of all pairings were found to exhibit synergy though this figure is skewed by Bodytrack where 35.3% of all pairings were classified as having a synergistic interaction.

In total, we found 38.5% of modification pairings produced antagonistic behaviour. To obtain the most optimal solutions, it is evident that effective modifications must be selected carefully and therefore greedy approaches will rarely produce superior solutions. Though 61.5% of modification pairs were worth combining, the rate of antagonism is high. For this reason, we advocate the

usage of evolutionary search techniques such as GAs. They are capable of building complex solutions by testing the interaction of components. Bad interactions incur a fitness penalty thereby disincentivising their combination in the population even when individually they are of benefit.

We chose to investigate an instance of synergy and an instance of antagonism to better understand the phenomenon. 7zip was found to have one instance of 'synergy' and was the first found in this investigation. We therefore dedicated some time to investigating the synergistic interaction.

The two modifications responsible for synergy in the case of 7zip altered lines in two separate classes, `LzmaEnc.c` (shown in Figure 3.12) and `LzFind.c` (shown in Figure 3.13). We monitored the control flow of the `LzmaEnc.c` class when running without any modification, modification just in `LzmaEnc.c`, modification just in `LzFind.c`, and, finally, when both classes were modified. We observed that the control flow in `LzmaEnc.c` was the same whether the `LzFind.c` modification was applied exclusively or no modification was applied. When the `LzmaEnc.c` modification was solely applied it resulted in the skipping of a large portion of a frequently iterated for-loop. This is shown Figure 3.12 where line 40 is deleted, thereby leaving `numAvailFull` uninitialised. In our analysis this resulted in the `IF` condition at line 46 being true at a much higher frequency than it would when `numAvailFull` was initialised. When both modifications were present, the `LzmaEnc.c` maintained this behaviour but, in addition, the likelihood of returning at Line 35 also increased.

The modification in `LzFind.c`, in Figure 3.13, achieved this additional, synergistic behaviour. This modification, the application of a *delete* operation at line 40, turned `while (++len != lenLimit)` to `while (false)` in the `Hc_GetMatchesSpec` method. We found this resulted in the value returned by `Hc_GetMatchesSpec` having a higher likelihood of being a lower value. This is passed, via `GetMatches`, to the `ReadMatchDistance` method in `LzmaEnc.c`. This value is then used to calculate a variable, `lenRes`, the value of which `ReadMatchDistance` eventually returns and assigns to `lenEnd` in `getOptimum` at line 26. Though other factors feed into this calculation, a lower value returned by `GetMatches` results in a lower value returned by `ReadMatchDistances`. Ultimately, the smaller the value of `lenEnd` the quicker the return statement is encountered at Line 35.

We found this modification reduced the execution frequency of the more expensive false branch of the `IF` statement by 15.7%. Individually, the `LzmaEnc.c` modification achieved a 24.7% reduction in energy consumption. Likewise, in the case of the `LzFind.c` modification a 17.5% reduction in energy reduction was found. When both were combined a 43.4% reduction was achieved. The synergistic effect resulted in an 'additional' saving of 1.2%.

In a similar vein, we investigated an antagonistic reaction. In Bodytrack, we found two modifications that both deleted parameter declarations in Bodytrack's `CameraModel.c` class. One deleted `mc_ext(1,1) = Rc_ext[0,1]` and another deleted `mc_ext(1,2) = Rc_ext[1,2]`. Known as the 3D Displacement Matrix, `mc_ext` is utilised frequently in Bodytrack. In both cases, when these matrix values were left undefined, energy consumption reduced. When applied together, a smaller energy consumption reduction was measured than when either were applied individually. We cannot fully explain this effect as both modifications work by leaving these matrix values as uninitialised; undefined behaviour in C. However, the effect on energy consumption differed depending on whether both values were uninitialised or only one was. We found that application of the first *delete* operation alone resulted in a 17.6% reduction in energy consumption. The second *delete* operation, when applied alone, reduced energy consumption by 13.8%. However, when applied simultaneously, we found energy reduced by only 11.6%, lower than when either were applied individually.

3.3.6 Discussion

In RQ1, we showed that the measurement framework we used in this investigation was sufficient to understand the energy optimisation search space in GI. However, we showed that our energy measurement framework did not produce reliable results across devices. We also observed that, in line with observations from other researchers [98], node restarts can affect energy readings. Despite this, we showed that *proportional* differences in energy readings between devices are consistent, and believe it is important that those working in energy optimisation research are aware of these issues. The seemingly simple task of being able to measure what is being optimised remains a significant hurdle in energy optimisation research [34] and, thus, care and consideration is needed when planning experiments. The Raspberry Pi/MAGEEC board setup is an abstract representation of 'real-world' systems, which researchers can use to gain insight into energy consumption in a manner which is controlled, low cost, and easily expandable. We acknowledge that more accurate results may be obtained with more expensive devices, though this would come at the cost of having less devices running in parallel, limiting the amount of data that may be gathered in a given timescale. We believe the setup we have used could be utilised in a variety of other energy optimisation work.

With this energy measurement setup, we analysed what was possible with the application of a single modification in RQ2. We conclude from this data that the *delete*, *copy*, and *replace* operators are largely ineffective at optimising energy consumption with only 0.09% capable of producing a statistically significant reduction in energy consumption while preserving output quality in GI. Evaluations of code modifications are typically costly, as checks must be done to ensure functionality has been preserved. On top of this, testing must determine the modifications' effect on the target property (in our case, energy consumption). We have shown that less than one in every thousand modifications is effective. Though evaluations of ineffective modifications is expected in GI, this rate is extremely high. Some practitioners of GI who target energy optimisation have had success with more bespoke operators, such as swapping of Java Collection implementations [121], or alteration of colours in GUI interfaces to reduce the energy consumption of OLED Smartphone screens [116]. We argue this is a good avenue for research as our findings suggests the 'standard set' of operators discussed in this paper (i.e., *delete*, *copy*, and *replace* at the source code line level) are not very effective.

We have shown that permitting a degradation in output quality can aid in the optimisation of energy efficiency. We observe that the number of modifications that can reduce energy consumption increase from 0.09% to 1.36% when approximation is permitted; a 15-fold improvement in the number of effective modifications. We appreciate that a lot of these approximate solutions are undesirable and that, in most cases, there are limits on how approximate an output can be. However, we can find no evidence that permitting approximation will impede developer's ability to reduce energy consumption. Many multi-objective optimisation methods are available [124] and have already seen adoption in genetic improvement research [41, 40, 116, 184]. We believe integrating 'output quality' as an objective can significantly benefit future projects.

In RQ3, we explored the wider search space by analysing the effectiveness of combining modifications. Our analysis shows that both synergy and antagonism are present in the search space. If there was low antagonism, we could advocate a greedy approach as the combination of any effective two modifications rarely produced a non-linear negative effect. However, we did not observe this. In fact, 38.5% of modifications produced some form of antagonism. For this reason, we advise more advanced search techniques such as genetic algorithms, though any search that effectively tests combinations of modifications would be sufficient. Simply combining any and all effective modifications will not produce the sums of their parts in all cases.

In answering RQ1, we showed that, with careful analysis and understanding, we can report reliable results. We could have reduced variance more by running applications on a bare machine, thereby removing interference that may emerge from the operating system. This may produce results of greater interest as running on a bare machine is more common in embedded systems which, unlike the Raspberry Pi devices studied, may be battery powered, making the goal of reducing energy consumption more important. The 'Internet of Things' is likely to constitute of many small embedded systems powered by batteries that are expected to run for a certain time before depletion. It would therefore be useful to explore this area in future; however, in this work, we focused on a more typical architecture — that with an application working on top of an operating system. The primary motivation for this is that there is considerably more open-source software available for optimisation in such an environment.

Our measurement cluster can expand indefinitely, thus allowing more modifications to be evaluated in parallel. We have explored a very small area of each respective search space. In future work, it may be of value to explore the wider search space (i.e. interaction between many modifications).

3.3.7 Threats to Validity

The work presented here uses direct energy measurements. Though this results in more reliable evaluations compared to the ones based on simulation, these direct energy measurements inevitably also contain variance. We have quantified this in answering RQ1 but it means that modifications which produce very small but positive changes are undetectable. While we detected energy decreases as small as 0.009%, there may be modifications that produce even smaller changes that were simply undetectable with our framework. Our investigations, however, show that modifications which produce detectable, non-trivial, energy reductions are rare.

In the investigation outlined in this section, we have been careful to ensure that any modifications reported as being effective truly are. To achieve this aim, our requirements for what constitutes an 'effective modification' have been strict. For a modification to be classified as effective, it must produce a solution in which we observe a statistically significant decrease for all test cases. While we believe this to be the most honest approach to presenting the data, it may not be representative of real-world genetic improvement where modifications can be seen as effective if they cause improvements in only a proportion of test cases. Determining at what point we may classify a modification as effective is subjective and thereby left to the GI practitioner's discretion. We have chosen to be strict rather than risk being too lenient, thereby avoiding the presentation of results that may not be applicable to all those in the GI research community.

All the applications we have chosen to study have user-level parameters which may be modified. Some of these parameters may be used to further approximate output quality while reducing energy consumption. We have not experimented with traditional application parameter tuning and have therefore not made comparisons between the results presented here and what may be achievable through other techniques. Such a study is outside of this investigation's scope but we acknowledge there are other established methods to trading application output quality for reductions in energy consumption.

As we only target Raspberry Pi devices running the Raspbian OS, we cannot ensure that the conclusions drawn from this investigation are universal across all software systems. We acknowledge that results are likely to be different when investigating hardware that utilises complex I/O components such as wireless network interfaces which are known to consume large amount of energy in mobile devices [114]. More research will be needed to investigate such platform and system-specific

variations.

We chose to investigate the *copy*, *delete* and *replace* search operators because of their frequent use in state-of-the-art genetic improvement work [108, 109, 145]. Other search operators, such those associated with Deep Parameter Optimisation [184] or API level changes [121], may function better in the context of energy consumption; however, our aim was to investigate the nature of the search space produced in modern GI research.

```
1  static UInt32 ReadMatchDistances(CLzmaEnc *p,
2    UInt32 *numDistancePairsRes)
3  {
4    UInt32 lenRes = 0, numPairs;
5
6    ...
7
8    numPairs = p->matchFinder.GetMatches(
9      p->matchFinderObj, p->matches);
10
11     /*
12      * Calculation determining
13      * the value of lenRes
14      */
15
16   ...
17
18   return lenRes;
19 }
20
21 static UInt32 GetOptimum(CLzmaEnc *p,
22   UInt32 position, UInt32 *backRes)
23 {
24   ...
25
26   lenEnd = ReadMatchDistances(p, &numPairs);
27
28   ...
29
30   for(;;){
31     UInt32 numAvailFul;
32
33     cur++;
34     if (cur == lenEnd){
35       return Backward(p, backRes, cur);
36     }
37
38   ...
39
40     numAvailFull = p->numAvail; //Mod: Delete
41     UInt32 temp = kNumOpts - 1 - cur;
42     if (temp < numAvailFull){
43       numAvailFull = temp;
44     }
45
46     if (numAvailFull < 2)
47       continue;
48
49   ...
50
51   }
52 }
```

Figure 3.12: Synergy Modification (`LzmaEnc.c`).

```
1  UInt32 GetMatches(CMatchFinder *p,
2    UInt32 *distances)
3  {
4    UInt32 offset;
5    curMatch = p->hash[hashValue];
6    offset = (UInt32)
7      (Hc_GetMatchesSpec(lenLimit,
8      p->hash[hashValue],
9      MF_PARAMS(p),
10     distances, 2) - (distances));
11   return offset;
12 }
13
14 static UInt32 * Hc_GetMatchesSpec(
15   UInt32 lenLimit,
16   UInt32 curMatch, UInt32 pos,
17   const Byte *cur, CLzRef *son,
18   UInt32 _cyclicBufferPos,
19   UInt32 _cyclicBufferSize,
20   UInt32 cutValue,
21   UInt32 *distances, UInt32 maxLen)
22 {
23
24   ...
25
26   for (;;)
27   {
28     UInt32 delta = pos - curMatch;
29     if (cutValue-- == 0
30         || delta >= _cyclicBufferSize)
31       return distances;
32     const Byte *pb = cur - delta;
33     curMatch = son[_cyclicBufferPos
34          - delta
35          + ((delta > _cyclicBufferPos)
36          ? _cyclicBufferSize : 0)];
37     if (pb[maxLen] == cur[maxLen] && *pb == *cur)
38     {
39       UInt32 len = 0;
40       while (++len != lenLimit) //Mod: Delete
41         if (pb[len] != cur[len])
42           break;
43       if (maxLen < len)
44       {
45         *distances++ = maxLen = len;
46         *distances++ = delta - 1;
47         if (len == lenLimit)
48           return distances;
49       }
50     }
51   }
52
53   ...
54
55 }
```

Figure 3.13: Synergy Modification (`LzFind.c`).

Chapter 4

Approximate Computing

4.1 Introduction

In Chapter 3, we outlined investigations into using genetic improvement to reduce software's energy consumption. The chapter ended by telling of an investigation which, in part, explored the usage of approximate oracles to find trade-offs between energy consumption and software correctness. This is part of an area of software engineering known as *Approximate Computing*, where software's output may be approximated if it produces beneficial results in other properties. Complimenting our work, Chippa et al. carried out an investigation into applications' resilience to approximation [52] and found applications often contain modules that are resilient to approximation, and that these modules take up a large fraction of execution time (67% to 96% in the applications they studied). Given our own previous research into the energy-approximation search space, and the impressive results obtained by other researchers, we decided to dedicate some time to the area of approximate computing and how GI may be used to find useful approximations. We start with Section 4.2 which outlines an investigation into how execution time was reduced in the Viola-Jones algorithm by reducing its accuracy. We did this by using a very specific form of genetic improvement known as *Deep Parameter Optimisation*, which we found was an effective approach to achieve approximation. From this we moved onto the work outlined in Section 4.3 which attempted to tackle the domain of mobile devices with the goal of reducing energy consumed while approximating the output of a commonly used Android library. Though the investigation outlined in Section 4.3 was littered with difficulties, we none-the-less report it as a proof-of-concept of optimising energy consumption using deep parameter optimisation in mobile devices.

4.2 Execution time vs. Function correctness

OpenCV is a computer vision library containing a wide variety of algorithms, commonly used by the AI community. These algorithms are frequently deployed to real-world environments in which execution time is a critical factor. AI libraries, such as OpenCV, provide algorithms which have been developed to maximise accuracy, normally at the cost of higher computation time. However,

in many domains such high accuracy may not be noticed and the cost to achieve such accuracy unnecessary, if not intolerable. For example, when a drone is performing a visual survey to detect objects of interest [78], is imperative the drone's deployment time is kept to a minimum due to its limited energy budget. In such domains, having a small inaccuracy in detection may be tolerable if it results in the task being completed within an acceptable timeframe. Naturally, what trade-off is required is dependent on circumstance. In some environments, high accuracy is needed and quick execution less so, though in other environments the opposite is true. It is for this reason we wish to develop techniques that present developers with Pareto frontiers of applications to choose from.

In this Section we outline an investigation in which we optimised a face detection algorithm, implemented using OpenCV, with a genetic improvement technique known as deep parameter optimisation [184]. We created a Pareto frontier of applications, trading execution time and classification accuracy, for a user to select from and deploy as needed. This investigation set out to be a proof-of-concept for those who wish to create approximated programs using genetic improvement.

4.2.1 OpenCV

OpenCV is a library for computer vision [38]. It was developed by Intel and has had wide adoption in the computer vision community, as well as the AI community in general. Face detection in OpenCV is commonly implemented using the Viola-Jones algorithm [170, 171]. The Viola-Jones algorithm searches an image, at multiple scales, shifting through the image one pixel at a time, for a collection of Haar features — shapes of binary values defining areas of light and darkness. The selected set of Haar features defines what the algorithm is capable of detecting. OpenCV comes with a set of Haar features that detects human faces, which we use in this investigation, though a classifier for other objects can be generated with an appropriate training set.

4.2.2 Deep Parameter Optimisation

Deep parameter optimisation [184] is a genetic improvement technique where the source code operators function by toggling, so called, 'deep parameters' found within its source code. What constitutes a 'deep parameter' is anything within code, not previously exposed to the user, which may exist in multiple known forms while preserving some fundamental functionality. For example, in Java, when a developer wishes to use a collection they must declare which subclass of the abstract superclass `java.util.Collection` to implement (e.g. `java.util.ArrayList`, `java.util.HashSet`, etc.). It has been shown that these different subclasses have different non-functional properties, such as energy consumption [141]. In the vast majority of cases the developer will choose an implementation based on his own intuition or preferences; utilising little information on how this may effect the software system's performance. In essence, the developer hard-codes these parameters based on feeble assumptions. In deep parameter optimisation we expose these parameters to be toggled and optimised using a search-based approach.

Previous work by Wu et al. used deep parameter optimisation to find Pareto frontiers of the standard C library's `malloc` function [184]. In their work they did not consider functional properties to be a valid target property, and instead aimed to find trade-offs between execution time and memory consumption. They were capable of finding solutions that improved execution time by up to 12%, or reduced memory consumption by up to 21%. To the best of our knowledge we were the first to use deep parameter optimisation in the domain of approximate computing.

4.2.3 Experimental Setup

Given OpenCV is a library, we developed a small command-line level program to utilise the OpenCV functionality we wished to optimise. This program, `classify_images`, took a directory of images as a lone argument. When executed `classify_images` produced an output identifying which images in the directory contained faces and which did not. `classify_images` utilised OpenCV's `CascadeClassifier::detectMultiScale` method with `CascadeClassifier` initialised using `haarcascade_frontalface_alt.xml` — the classifier for detecting faces, included by default in the OpenCV library.

We created a set of 20,000 images. Half of the images contained faces, which we obtained from the University of Massachusetts's 'Labelled Face In the Wild' dataset [94]. The other half were images which did not contain faces and were obtained from California Institute of Technology's 'Caltech-256' dataset [80]. This set of 20,000 images was then split into a training set containing 1,500 images with faces and 1,500 without, and a test set containing 8,500 images with faces and 8,500 without. Prior to any form of optimisation `classify_images` incorrectly classified 0.90% of the training set (taking 34.76 seconds to process) and 1.04% of the test set (taking 191.27 seconds to process).

We profiled the software to find which files were the most heavily utilised in OpenCV when classifying images. We found that the top two files were `cascadedetect.cpp` and `cascadedetect.hpp`. We then proceeded to extract all integer constants from these files. This process involved using a regular expression to highlight all instances of integer constants. Prior to doing so we replaced all occurrences of `[variable]++` to `[variable]+=1` and all occurrences of `[variable]-` to `[variable]-=1`, thereby increasing the number of integer constants available for extraction.

We then replaced all instances of integer constants found with distinct C++ define precompilation macros. These were extracted to a file called `defines.hpp` which was then included (via `#include`) in both `cascadedetect.cpp` and `cascadedetect.hpp`. In total, `defines.hpp` contained 537 integer constants. `defines.hpp` can be seen as a source code level configuration which declares the values of the integer constants. It is these values we modified in our investigation.

At this stage we believed having 537 integer constants to toggle was excessive as, in all likelihood, many of these parameters would not be worth toggling. We believed we could effectively categorise these exposed parameters into three separate categories: those that were too sensitive (i.e. small changes in their values resulted in crashes or time outs), those that were too insensitive (i.e. large changes in their value did not have any effect on execution time), and those which existed somewhere between these extremes which we believed worthy of optimisation. To do so, for each parameter we first incremented its value by one, compiled `classify_images` and ran it against a single image randomly select from the training set. If `classify_images` crashed, or the classification of the selected image was incorrect, then this parameter was classified as 'too sensitive'. Otherwise, the parameter's value was incremented by 50 with `classify_images` recompiled and run against the entire training set, measuring its execution time. If the measured execution time was within the 95% confidence interval for the original, unmodified, `classify_images`, run 100 times on the training set (we did not consider output quality during this step), then the parameter was classified as 'too insensitive', otherwise the remaining parameters were deemed 'modifiable' for the needs of our experiment. After these steps we were left with 51 deep parameters that were tagged as 'modifiable', which we targeted for modification in this investigation

With this in mind, we represented our program as a tuple of integers which we could then alter to produce software variants. We hypothesised that by modifying this simple representation, the corresponding changes to the source code could result in useful approximate variants of the software.

For example, in C++ `FOR` structures such as `for(int i=0; i<50; i+=1){ ... }`, it is easy to see how loops (which are often costly) maybe manipulated through the toggling of the integer constants. The number of iterations may be reduced, a technique known to produce approximated outputs while reducing execution time [156], or the body may be disabled entirely, thus removing a potentially costly branch from execution; a common way for deep parameter optimisation to reduce execution time as observed by Wu et al. [184].

Given our representation, we decided an ideal approach to optimisation would be using a genetic algorithm. As we wished to optimise for two objectives we utilised NSGA-II [58], a genetic algorithm designed for multi-objective search. We chose to use NSGA-II as implemented by the MOEA Framework [7], version 2.9. We setup the MOEA framework with a population size of 100 and a maximum number of evaluations of 10,000 (i.e., 100 generations). For MOEA to function the fitness functions of the objectives must be provided. In this investigation these were the execution time of `classify_images` and its classification inaccuracy on the training set. NSGA-II attempted to minimise both of these objectives. To measure the execution time objective we used UNIX's `time` utility on `classify_images` when classifying the training set. The classification inaccuracy was calculated as a percentage of incorrect classifications by `classify_images` on the training set. As an integer in C++ may have 2^{16} unique values, we reduced the search space by only allowing the integers to be increased to a maximum of 64 and to be decreased to a minimum of 0.

The procedure for creating the initial generation is shown in Algorithm 7. The algorithm starts by creating a population P containing S vectors. Each vector V contains N elements $(v_1 \dots v_N)$, each of which is a natural number ≤ 64. Every element within the vector represents one of the N deep parameters — i.e., v_i would represent the i^{th} deep parameter. To start with, each vector in the population is set to the software's initial state. The first vector in the population, V_1, maintains the original state of the program. From the second vector onwards ($V_2 ... V_S$), each is modified to produce a solution neighbouring the original. This works by taking the n^{th} element in the vector (starting at the first), and incrementing it by x, where x is the number of iterations through the entire vector length thus far.

We ran this NSGA-II setup in an Ubuntu 14.04.4 m4.large Amazon EC2 Instance (2x2.4GHz Intel Xeon E5-2676 v3 processor, 8 GiB of memory, SSD Storage).

Algorithm 7 The initial population generation algorithm.

Require: N, number of deep parameters, $N \in \mathbb{N}$
S, population size, $S \in \mathbb{N}$

1: $V = \langle v_1 \dots v_N \rangle, (\forall v \in V)[v \in \mathbb{N} \wedge v \leq 64]$
2: $P = \langle V_1 \dots V_S \rangle$ # The population
3: $\forall V_i \in P$: Set V_i be equal to the software's initial state
4: **for** $i \in (2 \dots S)$ **do**
5: $\quad n \leftarrow (i-1) \mod N$
6: $\quad x \leftarrow \lfloor \frac{i-2}{N} \rfloor + 1$
7: $\quad [V_i]_n \leftarrow [V_i]_n + x$
8: **end for**

Once MOEA completed execution it returned the Pareto frontier of solutions. To check these results were not over-fit to the training set, we ran each Pareto optimal solution on the test set, removing any which crashed or were dominated by other solutions to produce the final Pareto optimal set.

4.2.4 Introduction of a bug

Though unbeknownst to us at the time of carrying out our experiments, our setup actually contained a bug. As already stated, our setup involved seeding the initial generation with a single instance of the original solution (that is, all the deep parameters as they appear in the original, unmodified source code) and close variants. Though this did occur, there was an error in our implementation that resulted in this seeding procedure being run for every generation, thus overwriting any alternative generation produced by the MOEA framework. However, the MOEA framework maintains a separate set of all the Pareto optimal solutions found (a form of elitism), a Pareto frontier was ultimately produced.

The bug was equivalent to running the initialisation algorithm for each generation and changing Line 6 in Algorithm 7 to '$x \leftarrow \lfloor \frac{i-2+gN}{N} \rfloor + 1$' where g is the current generation — $g \in \{\mathbb{N} : 0 \dots G\}$ where G is the number of generations. The initial generation is considered as generation zero, $g \leftarrow 0$. The actual buggy Java code and fixed implementation can be found in Appendix B.

This unintentionally resulted in a search algorithm that *systematically explored the local search space rather than explored it in an evolutionary manner* as we had originally intended. We decided to keep the data, produced by the buggy run, and discuss it in Section 4.2.5. We then fixed the bug and ran the setup again. These results are discussed in Section 4.2.6.

We refer to these 'versions' of the code as the *Systematic* and *Evolutionary* implementations for the 'buggy' and 'correct' implementations respectively.

4.2.5 Results (Systematic Implementation)

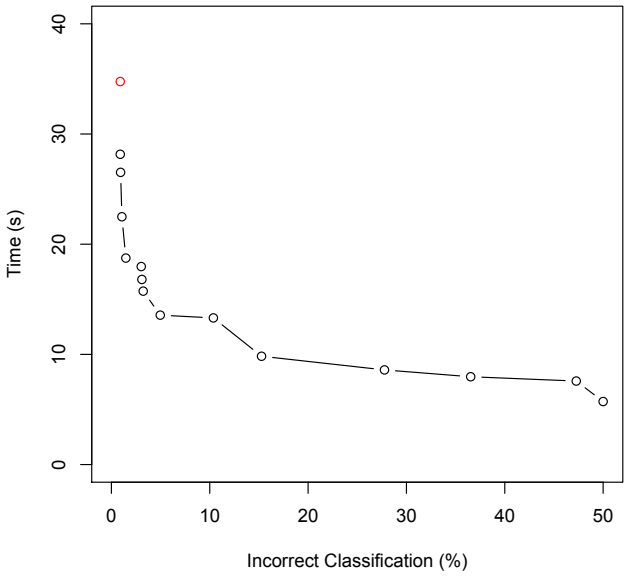
When running the systematic implementation, the NSGA-II algorithm produced a Pareto frontier that contained 14 solutions when run on the training set. This is shown in Figure 4.1a.

We then ran each of these Pareto optimal solutions on the test set. One failed to cease execution and another was dominated by other solutions in the set thus leaving 12 Pareto optimal solutions. These are shown in Figure 4.1b (the original program included as the top left solution). As can be seen, the Pareto frontier largely holds its performance on the test set, showing the results have not succumb to overfitting.

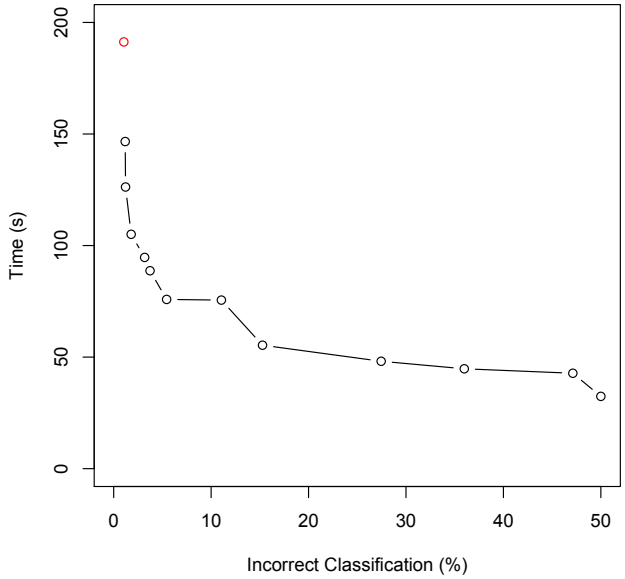
On the test set, the original solution ran in 191.27 seconds and incorrectly classified 1.04% of solutions. The original solution is Pareto optimal, and the next solution on the frontier runs in 146.62 seconds (a 23.34% reduction) with 1.18% images incorrectly classified. We can then go down the frontier to a solution which runs in 75.84 seconds (a 60.35% reduction) with a classification inaccuracy of 5.44%. The final solution on the frontier runs in 32.40 seconds but has a classification inaccuracy of 50%. As our applications are 50% faces, and 50% non-faces, an inaccuracy rate of 50% is a worst-case. We observed all this solution was doing was classifying everything as not containing a face.

4.2.6 Results (Evolutionary Implementation)

After running the (faulty) systematic implementation, we found and fixed the bug, thus creating our evolutionary implementation, as we had originally intended it to be. After running with a population of 100 for 100 generations, we produced the Pareto frontier shown in Figure 4.2a. It contains 9 Pareto optimal solutions. At this stage we can see that the Pareto frontier is similar to that produced via the systematic approach.



(a) Training Set



(b) Test Set

Figure 4.1: The Systematic Implementation's Pareto frontiers.

We then ran each of these Pareto optimal solutions on the test set. We found all but 2 of the Pareto optimal solutions were either dominated, caused an error during execution, or timed out when run on the test set. Figure 4.2b shows the Pareto frontier left (we include the original solution as reference as the solution in the top-left).

Interestingly, this Pareto frontier does not look nearly as healthy as the one produced by the systematic approach. It contains two solutions, one which runs in 32.41 seconds (an 83.06% reduction when compared to the original solution) with a 48.94% classification inaccuracy. This is close to the 50% worst-case. The other solution achieves this worst-case with a classification inaccuracy of 50% and an execution time of 32.47 seconds.

4.2.7 Conclusions and Discussion

In Figure 4.3 we show a layover of the Pareto frontiers produced by both the systematic and evolutionary approaches on the training and test sets. If we focus on the performance on the training set, we can see that the evolutionary approach produces a better Pareto frontier, which would suggest the evolutionary approach is better. However, if we focus on the test set, we can see the systematic approach performs the best. When comparing these two graphs, it seems like we have a contradiction; the evolutionary approach appears to be better on the training set but performs poorly on the test set (when compared to the systematic approach).

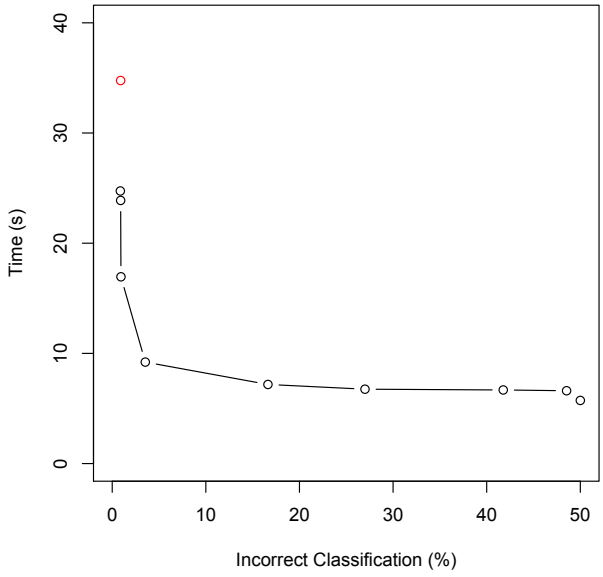
We believe this contradiction is due to the evolutionary approach overfitting to the training set. Whether this highlights a weakness in the evolutionary algorithm or a strength (it does perform very well on what it was trained on) is not for us to say, but since the approaches are ultimately evaluated on the held-out test set, we believe the systematic search worked best for this particular task. A better training set may have been able to fix this problem, or perhaps a validation set to stop the evolutionary algorithm from overfitting.

With a relatively simple setup and a budget of 10,000 evaluations, we have been able to produce a Pareto frontier of algorithms. Our intuition that a genetic improvement technique, such as deep parameter optimisation, can aid in the creation of these Pareto frontiers has been shown to be a correct one. We were capable of finding solutions that reduced execution time by over 60% while only increasing inaccuracy by a few percentage points. With some development, such technique could be of use to developers who wish to produce variants of the same algorithm for differing use-cases without significant manual effort.

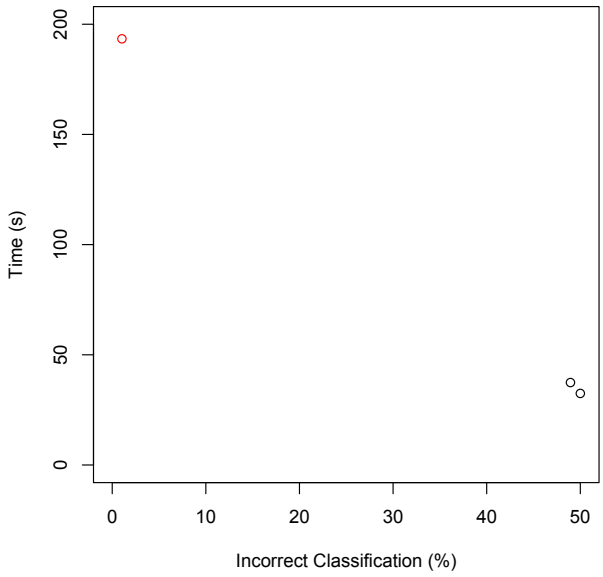
4.3 Energy consumption vs. Functional correctness

After carrying out the investigation into using deep parameter tuning we took time to reflect what we had found out thus far. We had shown that energy could be reduced, and that considerable savings could be achieved if a reduction in output quality was tolerable. We had also shown that deep parameter optimisation was an appropriate technique to achieve this aim. We therefore decided to combine this knowledge in an investigation into using deep parameter tuning to optimise the energy consumption of an application by allowing the output quality of the application to change.

We chose to target software in the mobile computing domain. Devices such as smartphones and tablets are a natural target for energy optimisation research. They are ubiquitous in modern society, yet are constrained by an energy budget; a constant annoyance for users of such devices. A survey of complaints made by users of Android applications found that having resource intensive features has a larger negative impact on an application's rating than uninteresting content or a

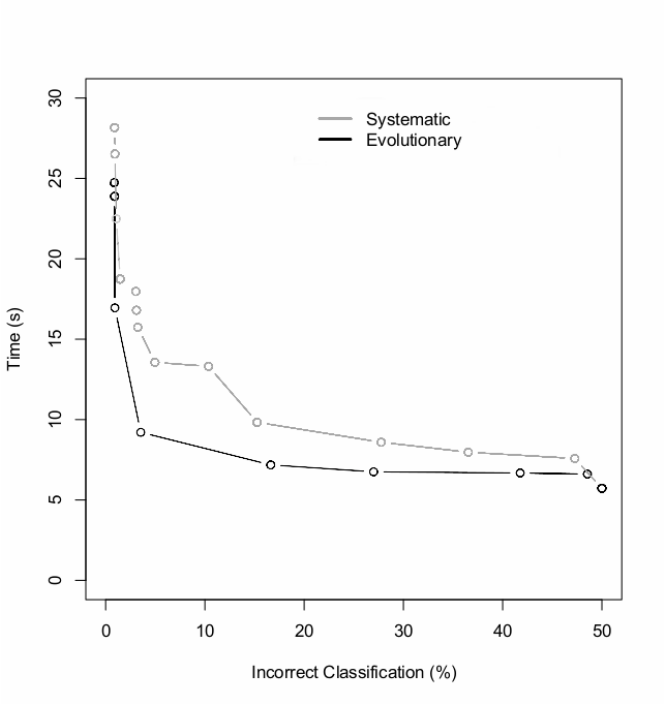


(a) Training Set

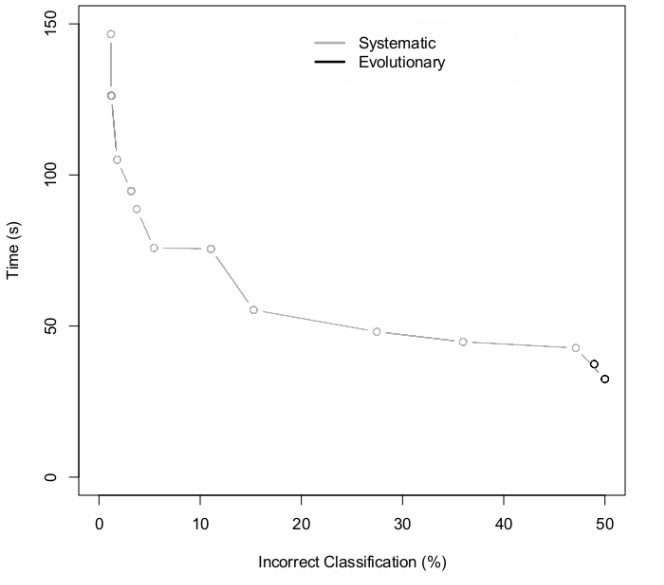


(b) Test Set

Figure 4.2: The Evolutionary Implementation's Pareto frontiers (original solution's performance included in the top-left).



(a) Training Set



(b) Test Set

Figure 4.3: The Evolutionary and Systematic Pareto frontiers compared.

poorly designed interface [101]. If one considers that the sale of smartphones now exceeds that of personal computers [49], and the average user spends in excess of 30 hours a month on smartphone applications [164], it becomes clear that research into energy optimisation on such devices is needed now more than ever.

In this section we outline research into reducing the energy consumption of *Rebound*, a Java physics library that models spring dynamics. It is used by popular Android applications like Evernote, Slingshot, LinkedIn and Facebook. During our experiments, we allowed for an approximation of the intended output, with the goal of finding a set of configurations that represented trade-offs between energy consumption and faithfulness to physically correct animations. In this investigation we encountered a number of difficulties: some expected, and others not. We documented these difficulties in this section and wish to frame this investigation as a 'proof-of-concept'; showing that energy optimisation is possible while acknowledging that much work is still required, particularly in obtaining reliable energy measurements.

4.3.1 Target Software: Java physics library Rebound

In order to select software to optimise, we first developed some selection criteria. We concluded the software must be widely used, computationally intensive enough so there is room for improvement, could be easily compiled within our setup, and that it should come with tests that allow for gradual deviations from the targets.

In searching for applications, we found the requirement that applications have tests that allow for gradual deviation to be the most troublesome. Tests tend to check the correctness of functional properties, but rarely provide any means in which we may derive a gradient of acceptability. Two notable groups of applications that did meet this criteria were internet browsers and media players, however, they were not considered due to the significant effort required in compiling them.

Following a comprehensive search for applications that satisfy all requirements, we settled on Rebound [11]. Rebound is a Java library that models spring dynamics. The spring models in Rebound can be used to create animations that feel natural by introducing real-world physics to applications (for example, in complex components like toggles, and scrollers). Major applications that use Rebound include Evernote, Slingshot, LinkedIn, and Facebook Home.

The target for our optimisation was the `Spring.java` class in Rebound's `com.facebook.rebound` package. This class implements a classical spring using Hooke's law with configurable friction and tension. Inside this class, the `advance` function is responsible for the physics simulation based on `SOLVER_TIMESTEP_SEC` sized chunks. The computations include, among others, Euler integrations and calculations of derivatives. Interestingly, some level of performance optimisation has already been performed, as evidenced by the source code comment:

> 'The math is inlined inside the loop since it made a huge performance impact when there are several springs being advanced.'

Rebound comes with 44 test cases. These tests vary significantly in nature. For example, some tests check if the ID of a spring is set correctly, and if Android listeners work as intended. Most importantly for us were the tests that perform the actual physics calculations. These calculations typically return an array, showing what position a spring element should be at each time-step, given a set of starting criteria. We found them to be relatively time consuming and could be interpreted in ways in which we could derive a gradient of output quality, both attributes we wished from our target application's tests.

In general, for a set of n tests with test inputs $T_1 \dots T_n$ and corresponding observed outputs $O_1, \dots O_n$, we measured the quality in three ways:

1. *M1*: How many tests were passed, as determined by `assertEquals(`T_i`,`O_i`)`, $1 \leq i \leq n$?
2. *M2*: If an array was produced, what was the average per-element deviation? We found variations here could result in unrealistic looking animations.
3. *M3*: If an array was produced, what was average array length deviation? We found variations here could result in animation that were too long or short.

Note that if for test i the output arrays of T_i and O_i differed in length, then *M2* considered only the first $min(|T_i|, |O_i|)$ fields.

Under the above testing regime the original code has the following outcome:

1. *M1*: All 44 tests are passed.
2. *M2*: The average deviation from the values provided by the oracle is zero.
3. *M3*: The average deviation from the oracle's array lengths is zero.

4.3.2 The Deep Parameter Optimisation Setup

Within this investigation we exposed both integer and double constants for optimisation via deep parameter optimisation (as opposed to just integer constants as in the previous investigation). To do this we started by replacing integer or double constants with placeholders. These placeholders referenced values that were read from a configuration at the start of the program's execution. In most genetic improvement research, modifications to the source code require recompilation before evaluation. This can be costly — within this investigation, recompilation carried a penalty of 20-30 seconds. With this setup the configuration file, which we may conceptualise as the exposed deep parameters, could be modified any number of times without recompilation. The configuration file was read once per execution and thus incurred a fixed energy overhead though. Since this was constant across any and all evaluations this did not affect the impact of our results.

In this investigation we started by profiling our target application, Rebound, and selected only those files which consumed large amounts of energy. In this case we found that most calculations were performed in just one Java class, `Spring.java`. The previously mentioned `advance` method, contained within `Spring.java`, performs the physics calculations, and is the second-most called method (9406 times, as determined by Cobertura 2.1.1, a Java coverage utility tool [2]). The most frequently called method is `isAtRest` (20340 times, also in `Spring.java`), which performs a rather simple check. All other methods did not have much impact on computation, and were only called a few dozen times, if at all. We therefore chose to target the `Spring.java` class exclusively as the remainder were deemed unlikely to contain parameters worth optimising.

Before exposing the parameters within `Spring.java` we, as before, replaced all instances of `{variable}++` with `{variable}+=1` and all instances of `{variable}-` with `{variable}-=1`. Once this is done we exposed 38 deep parameters. While one could begin tuning the parameters at this stage, it would be inefficient. Our previous work into deep parameter optimisation showed the majority of exposed parameters did not lead to significant improvements. We therefore categorised deep parameters as falling within three categories (as before) — too insensitive, too sensitive, and

those worth optimising. To do this, for each parameter, we incremented its value by 1 if it was an integer or by 10% if a double (all exposed doubles were non-zero). If this resulted in the program crashing when run we tagged this parameter as being too sensitive. Otherwise we multiplied its value by 10 (after the incrementation). If this resulted in a program in which energy consumption existed within the 95% confidence interval of the unmodified application's energy consumption (as determined by running the Rebound 100 times, measuring its energy each time) we tagged the parameter as being too insensitive. After these two steps were done for each parameter, we were left with a set of parameters which we concluded were worth optimising (i.e., neither too sensitive or too insensitive). In the case of `Spring.java`, we were left with 19 of the original 38 parameters after following the above steps.

As in our previous investigation, we represented our program as a tuple of values (integers and doubles). Again, we limited any parameter to have a minimum value of 0 and a maximum value of 64 (the parameters, when exposed, had initial values between 0 and 6). With a population size of 20, we seeded the initial generation with the original solution (i.e. the parameters as exposed from the initial, unmodified application) and those close to the initial solution in the search space (using Algorithm 7).

4.3.3 Target Hardware: Android Smartphones

In order to optimise mobile devices, one must take into account the specific hardware that is being targetted, how energy was measured, and how the software/OS running on the device was handled. We outline this here.

Hardware Platform

Modern mobile phones are equipped with battery fuel gauge chips that report the voltage, current and remaining energy within the battery [14]. The target devices for our experiments were the HTC Nexus 9 and the Motorola Nexus 6 (though, as we shall discuss later, we eventually dropped the Nexus 6 as the it was more susceptible to temperature-related energy reading variance). Both are equipped with the Maxim MAX17050 fuel gauge chip that attempts to compensate measurements against temperature, battery age and load [6].

For the optimisation of energy consumption, we solely relied on the energy readings as provided by the battery chip. This is in contrast to previous work (such as our work in Chapter 3) which relied on external meters, or software tools that estimated energy consumption. A brief practical characterisation of the internal battery meters on the Nexus 6 and Nexus 9 is included in Bokhari et al.'s work [35], in which the authors outline results for validating the precision of the internal meters under various workloads. In their work the internal meters were deemed sufficiently precise as long as the measurements were made over a non-trivial amount of time.

Software Framework

On the device we deployed a bespoke software framework. The framework's main job was to create test scenarios. It could activate, deactivate, and apply workloads to hardware components while keeping other components turned off, and the CPU at a fixed frequency. It also included a data logger and battery monitor. The data logger sampled hardware settings and utilisation data such as CPU frequency and load, screen brightness and network traffic.

The battery monitor recorded the power consumption data such as the remaining energy, voltage and drawn current during each test session. Accessing the battery chip's values was done through the battery API in Android's `BatteryManager` class. This API broadcasted these values with a frequency of 4Hz. We set out to gather as much data as was possible so we could alter our setup to reduce energy variance based on empirical evidence.

4.3.4 Getting the Experimental Setup Right - A Tale of Woe

Here, we report the various encountered challenges in developing our experimental setup. The discussion is divided into expected challenges and unexpected challenges.

Expected Challenges

System Behaviour

It is important to note that both considered devices are complex with many communication interfaces, controller chips, and multiple CPU cores, where much of the device behaviour is controlled by the operating system. The operating system, Android 6.0.1, is itself is complex, with system and user processes running in parallel with elaborate power consumption management in place.

To minimise the noise from these complex systems and to maximise the relative strength of the energy signal from our experiments it is important to reduce the energy footprint produced by background processes. To this end we deactivated communication interfaces using the flight mode. This prevented processes from transmitting data, which has been shown proven to substantially impact energy consumption, even when occurring in short bursts [33].

We also put the display in sleep mode, which reduces the power drawn for the display light and GPU. Turning the screen off allows the system to enter the so-called Doze-mode [15], which was introduced in Android 6, and which deactivates a number of system services that would otherwise inject noise into the energy signature of the experiments.

Another potential source of noise is the system dynamically adjusting CPU speed according the current workload. To avoid this issue we fixed the speed of all cores to the same value.

Sampling-Frequency-Induced Error

Some sampling error is induced by the fact that the battery fuel gauge can only be sampled with a maximum frequency of 4Hz. This means that some noise is introduced by gaps between the start and finishing time of the measured process and the time the battery is sampled. We minimised the impact of this by measuring the aggregate energy consumption of running each individual 20 consecutive times. This had the positive side effect of mitigating against background events which might infrequently occur (i.e. it is effectively averaged over 20 runs rather than skewing one measurement).

Unexpected Challenges

System Behaviour

There were a number of unanticipated challenges presented by system behaviour. One unexpected interaction stemmed from having the display in sleep mode which resulted in the system going into Doze-mode after the experiment starts. Doze-mode impacted the experiment by suspending

and rescheduling background processes including those we use to log data and run the Rebound library test suite. As a consequence, the test execution time increased drastically from seconds to hours in some cases. While the existence of Doze-Mode has benefits, it can be problematic in such experiments. To counter this problem we used partial wakelocks. Partial wakelocks keep processes running when, normally, they would enter Doze-Mode. In addition to this, temporarily activating the display after each generation of the evolutionary process allowed our framework to run Rebound's test cases normally. Needless to say, while the screen is on, the test execution was suspended.

Android Debug Bridge

The Android Debug Bridge (ADB) is a command-line tool by which developers can communicate with Android devices. It supports a variety of actions such as copying files to and from the device and installing/un-installing applications. The ADB consists of a client to initiate commands from the development machine, a daemon to execute command on the Android device, and a server to manage the communication between the client and the daemon.

In our framework, the ADB was used to deploy code on target devices. In preliminary runs the Rebound library was compiled, transferred to, and installed on the device for every change in its parameters. Unfortunately, these operations took up to a minute to complete, making iterative search impractically slow. Moreover, application transfer and installation could fail due to the ADB server's instability. Other failures included the connected device going off-line and other android services attempting to interface with the bridge.

To reduce deployment time the Rebound library was modified to read its code parameters from a configuration file (as explained in Section 4.3.2). This avoided the need to compile and transfer the application. To address the ADB connectivity issue a programmable USB hub was configured to automatically drop and restore the link to the device whenever the device went offline. The framework was also configured to poll for other services using the ADB and to restart the ADB server when this occurred.

Temperature

Temperature variations during experimental runs produced an unexpected source of systematic noise when attempting to run our experiments. In preliminary testing we observed that the battery temperature increased after successive runs of Rebound. We observed this temperature increase corresponded to an increased energy consumption. Figure 4.4 shows how the rise of battery temperature correlated with increasing energy consumption on the same program run 1000 consecutive times (100 trials of ten Rebound runs each) on the Nexus 6 device.

This observation is extremely problematic in the context of using genetic improvement. As the search is carried out we can assume the temperature will increase, which will therefore result in solutions found later in the search being declared as having a higher energy consumption than they would have if evaluated under earlier conditions. Genetic improvement depends on fair comparison between different software variants. Without this, GI is much harder.

To see if it was possible to learn the relationship between temperature and power consumption (and therefore compensate for it) we collected energy consumption for the same Rebound benchmark in varying temperatures.[1]. Figure 4.5 shows a scatter plot relating temperature to energy readings

[1]The experimental rig was placed in the refrigerator to extract some cooler readings.

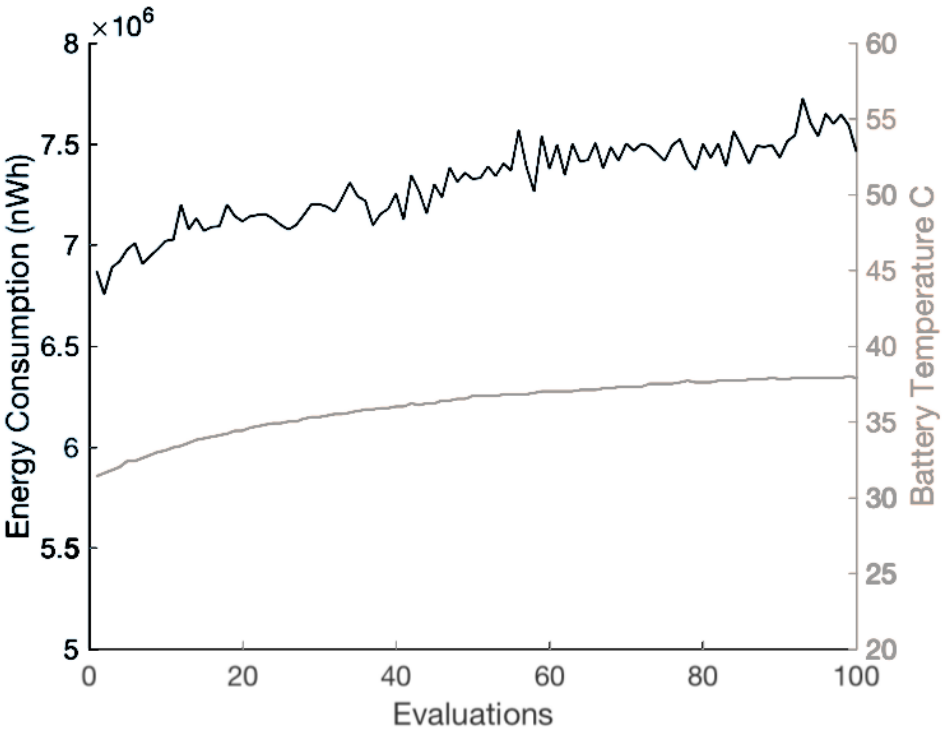


Figure 4.4: Battery temperature and average energy consumption.

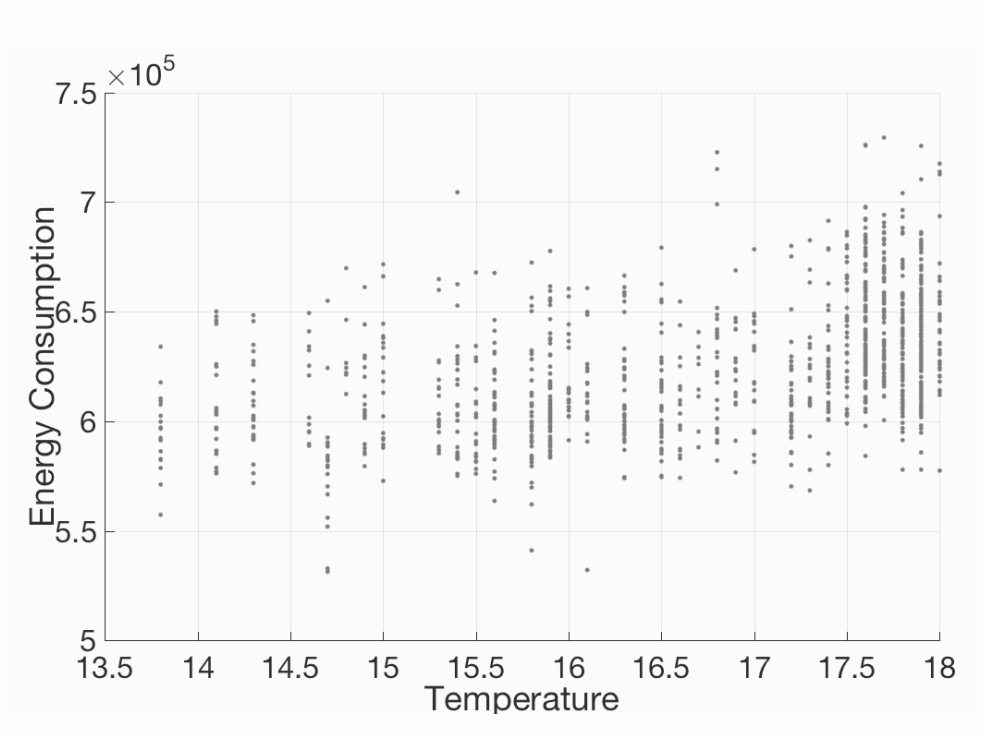


Figure 4.5: Scatter plot of temperature vs. energy consumption on the same benchmark.

from the same benchmark as previous. As can be seen, there is a general upward trend in the data.

We then ran non-linear regression on the resulting data using the GPTIPS2 [155] symbolic regression package. The learned function was: $E = 357T^2 - 6180.0T + 608000$ where E is the energy consumption of the benchmark in nWh and T is the temperature in degrees Celsius. The actual-vs-predicted curves for the sorted temperature data is given in Figure 4.6.

To avoid problems associated with temperature we switched to using exclusevly the Nexus 9 which exhibited less systematic variation in energy readings. However, given the difficulty in determining an accurate relationship between temperature and energy consumption, we decided instead to setup a test rig to control temperature by using a desk-fan coupled with sufficient egress for airflow around the device.

Processor Throttling

The test rig described above moderated rises in temperature but not enough to prevent a hardware CPU governor in the device from triggering reductions in processor speed. The effect of this fail-safe is shown in Figure 4.7.

The figure seems to indicate that throttling is activated by steep rises in temperature rather than high absolute values in temperature. The throttling also seemed to have the desired protective effect in slowing rises in temperature. Intermittent throttling is, of course, undesirable for experimental purposes as it introduces a variance between readings.

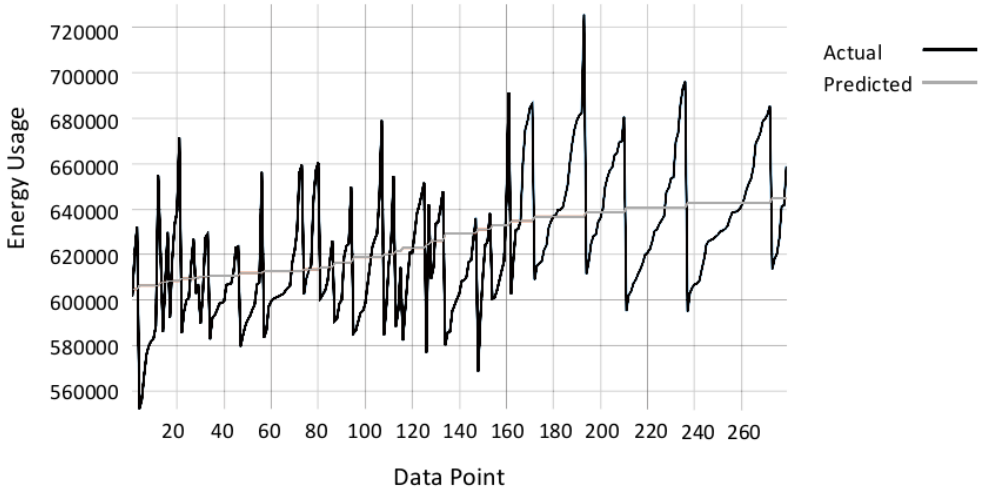


Figure 4.6: Actual energy consumption vs. predicted energy consumption as a function of temperature.

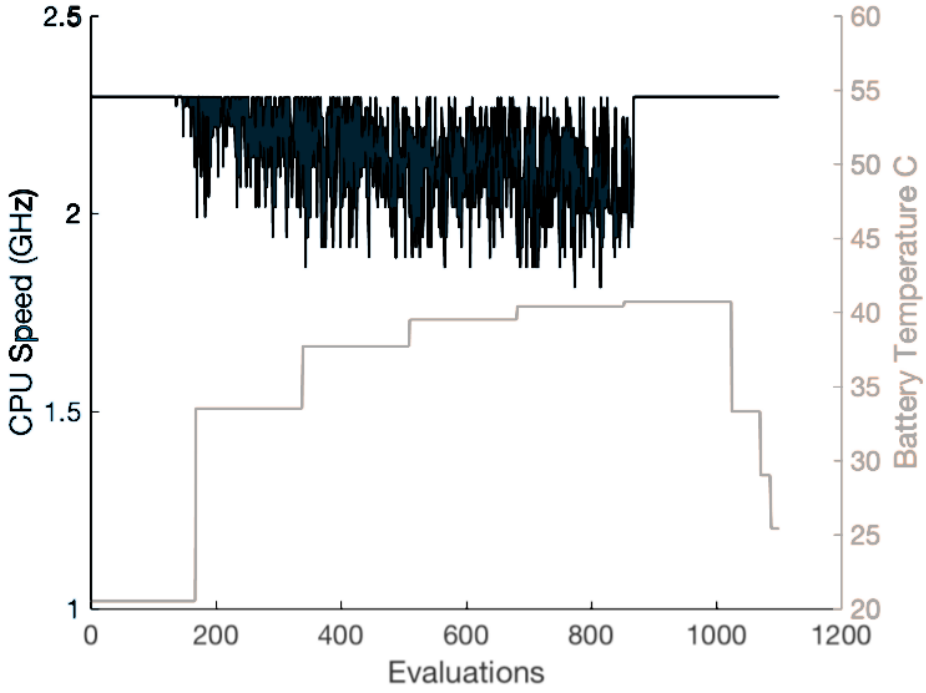


Figure 4.7: Battery temperature vs. processor speed over consecutive runs.

Informed by these results, we intentionally throttled processor speeds using the system governor to a fixed lower speed of 1.428GHz (as opposed to the Nexus 9's default 2.3GHz). We found a speed of 1.428GHz allowed the benchmark to run in a tolerable time-frame while reducing measurement variability and temperature increases.

Log Files and Memory

Further experiments with reduced processor speed revealed that per-run energy consumption still increased over multiple runs – albeit at a slower rate than before. Preliminary investigations revealed that the increase in energy consumption seemed to correlate with the size of logs and the size of the memory footprint of the Java test-harness used to run each generation. We re-designed the logging procedures to reduce the memory footprint of the generational log to 500kB. This reduction removed some of the variability in energy consumption but the increased energy consumption over multiple runs persisted. For further improvement we attempted to reduce the memory footprint of the test harness by calling the Java garbage collector (GC) every 250ms. This reduced the overall energy consumption level and growth. As an illustration of the impact of calling the GC, Figure 4.8 shows the sorted energy consumption data for a sequence of 100 runs with and without garbage collection.

After checking that both the with-GC and without-GC data were normally distributed (with a one-sample Kolmogorov-Smirnov test) we applied a t-test and confirmed that the with-GC runs were significantly less than the without GC runs with $p < 0.001$. It was later determined that the same effect could be achieved by calling GC after each generation.

4.3.5 Experiments and Results

After these issues were observed and rectified, we began to run our experiments. Here we report on our experiments in performing deep parameter optimisation of the Rebound Java physics library as described in the previous sections.

Evaluation times

Due to the communication and framework overheads, our evaluations were relatively time-consuming. 20 Rebound runs consumed a total runtime of approximately 30 seconds. However, the initialisation of the individual runs, and eventual clean-up by the framework, resulted in a total time of approximately 50 seconds per effective evaluation of a Rebound configuration on the device. Also, in order to allow for variation in Rebound's runtime, and to account for variations in logfile write times, sleeping threads, and other Android peculiarities, we set the timeout per configuration evaluation to two minutes.

As running tests consumes energy, we recharged the device after each generation. Interestingly, different USB cables result in different charging currents and thus in different charging times needed. For example, one cable's charging current was approximately 0.8A (as reported by the USB hub's software), whereas a 'better' cable allowed for 1.4A. Based on preliminary experiments, we conservatively chose a charging time of 20 minutes between generations.

The above meant that a single generation, with a population of 20 solutions, took approximately 1 hour on the device. While this seems costly, it is a big jump over some existing work, where a single configuration is evaluated by measuring the time needed for the device to run dry from 100% charge to 0% charge – which typically takes hours.

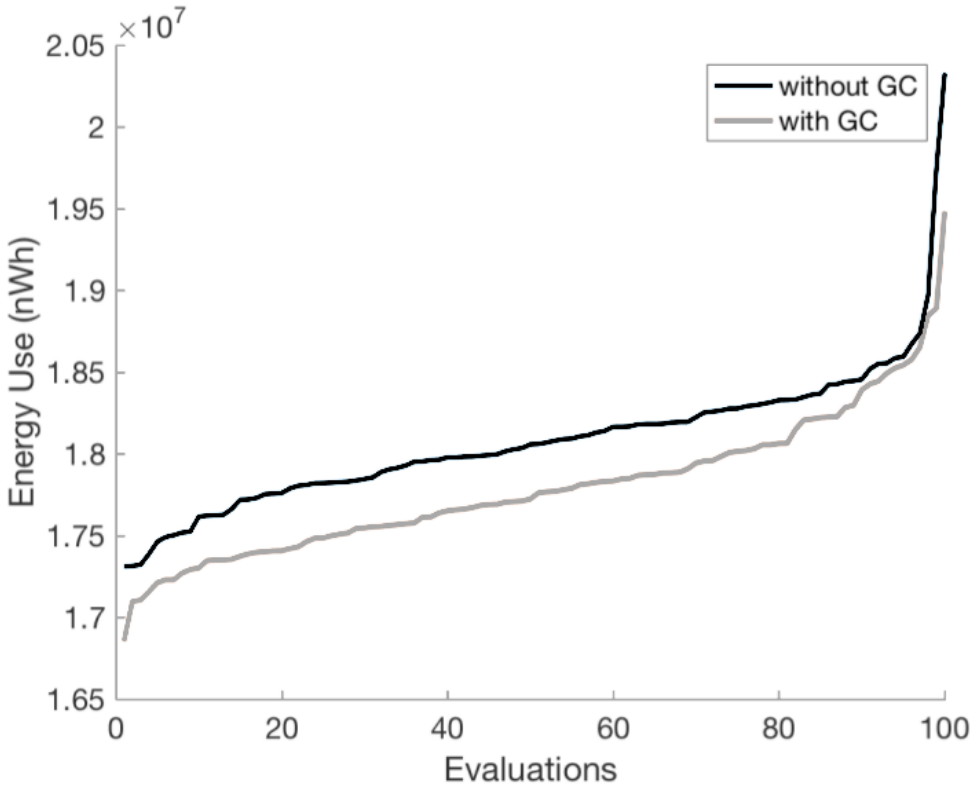


Figure 4.8: Sorted energy-use data with and without calls to the Garbage Collector.

Note that, as a welcome side-effect of our recharging strategy between generations, we can keep the battery close to its maximum charge. Within each generation, we observed only very minor drops in the battery voltage, for example, from 4.214V to 4.201V over a duration of approximately 15 minutes. This greatly reduces potential measurement drift by the fuel gauge chip.

Results

With this hefty evaluation time in mind, we ran our NSGA-II algorithm (implemented in the MOEA framework) for 340 evaluations. Each solution was evaluated by running against all 44 test cases. Our first objective was the minimisation of energy, and the second objective was the minimisation of deviation from the oracle's values. We also recorded the array length deviations *M3*, however, we only observed two cases: either the length deviation was zero, or the entire test timed out.

In our setup we extracted all solutions evaluated within the MOEA framework (regardless of whether they were Pareto optimal) for analysis. After removing duplicates and timed-out solutions, 55 evolved Rebound configurations remained that were different from the original one. Figure 4.9 shows these in the objective space, with the original configuration highlighted. Significantly, we can see solutions that consume less energy at the cost of decreased accuracy. In total, the 56 solutions achieved 43 different levels of accuracy. Their overall distribution of energy consumption covers quite a range, with the overall average being 37.5mWh and a standard deviation of 1.9mWh.

In terms of code features, it was difficult to gain consistent insights from the produced configurations, as we only had 19 decision variables, 44 test cases with physics simulations, and noise in energy and time measurements. Out of the 19 variables, nine had direct influence on the mathematics involved in the spring simulation. When these were changed, then the resulting calculations deviated from the oracle. Additional experiments would be needed to further reduce the noise in the energy and time measurements to better understand the influence of certain parameters on the algorithms behind the simulations and thus on the observed energy consumption.

Given the amount of technical difficulties described here, one might wonder if we could use a proxy function instead on mobile devices that did not have a dedicated battery chip installed. As one would expect, energy consumption and runtime (as reported by ADB) are correlated in our experiment, as we can see in Figure 4.10. However, this is just a moderate, positive correlation with a correlation coefficient $r = 0.7382$.

4.3.6 Discussion

As shown in this section, the improvement of software's deep parameters with modern mobile devices in-the-loop is feasible. However, our path to achieving this was littered with stumbling blocks, with some visible from afar and some only upon close scrutiny of results. Our main contribution from this investigation is highlighting to future researchers what challenges one might encounter when attempting to optimise energy on modern mobile devices.

Much like our previous research on MiniSAT (discussed in Chapter 3), it may be possible to optimise energy consumption using *estimates of energy consumption* rather than direct energy measurements. However, there exists no framework, at the time of writing, that is capable of giving reliable estimates of energy consumption across different hardware targets. Even if such estimates were reliable, their accuracy would not be known without a framework capable of calibrating and/or validating them. Thus, the need for reliable energy measurements is required first and foremost.

Though, setting up our energy measurement framework was not a trivial effort, we were capable of running a small 'proof-of-concept' that showed energy optimisation is possible in mobile devices

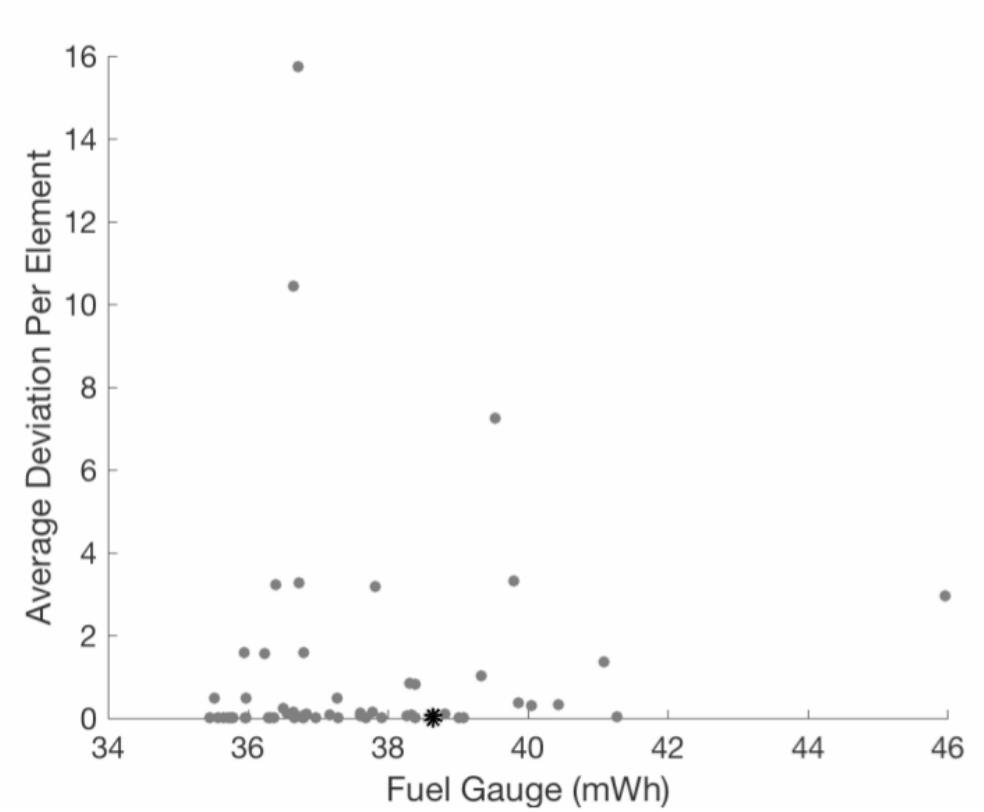


Figure 4.9: Scatter plot of energy vs. accuracy (the original solution has been highlighted).

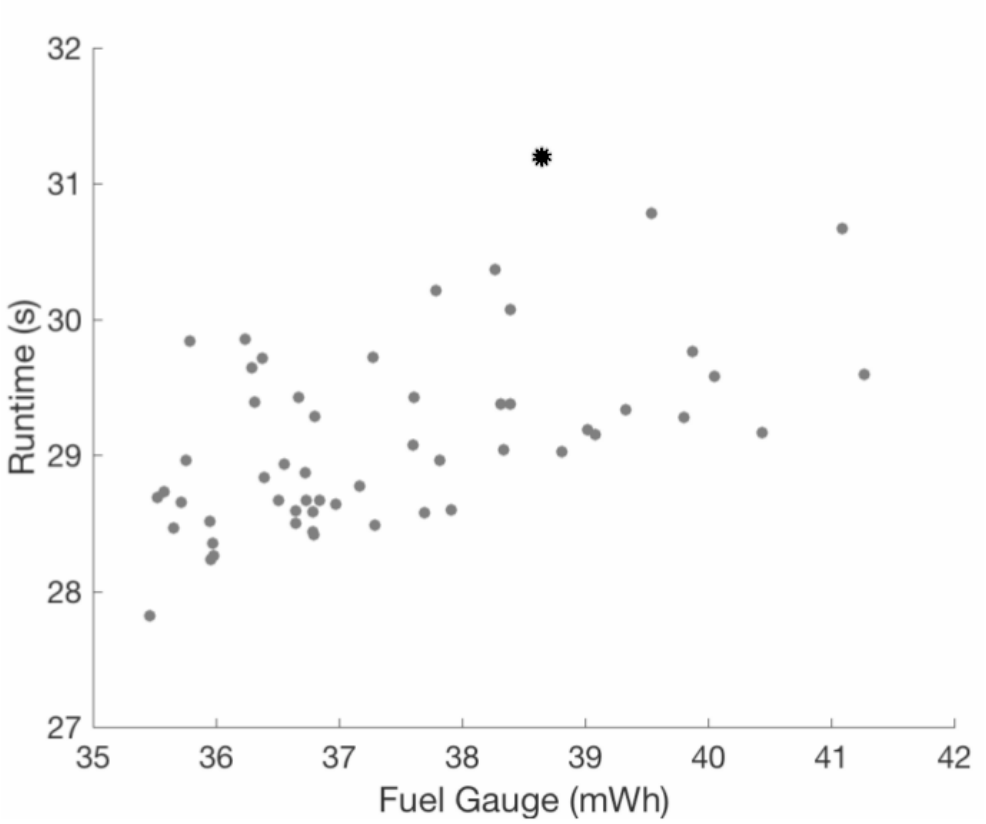


Figure 4.10: Scatter plot of energy vs. accuracy (the original solution has been highlighted).

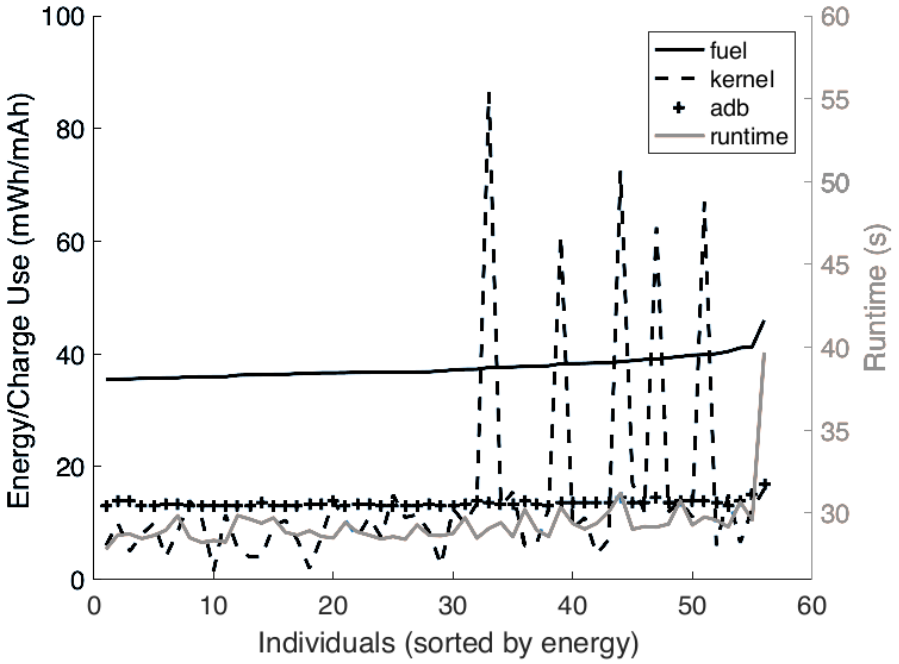


Figure 4.11: Correlation of energy with charge measurements and runtime.

using the 'deep parameter optimisation' GI technique. While the standard deviation across all readings was small, we were able to detect differences in energy consumption across different variants of the Rebound library, as well as changes in functional correctness.

Genetic improvement relies on being able to correctly classify solutions as being better, or worse, than others with respect to some objective. In the field of energy optimisation, this means having accurate and consistent energy measurements. We have shown in this investigation that measuring energy, and quality of output, is possible, and therefore, GI of energy consumption is possible within these energy-constrained mobile devices.

Despite this, we did not run a full GI investigation. The reason for this was primarily due to scalability problems. As there was variance, we were required to carry out many runs per evaluation in order to obtain an average. Furthermore, as we only had one device, we were incapable of running a significant number of generations in a reasonable timeframe. However, a parallel implementation of real-world evaluations across multiple modern mobile devices would be possible, much like the 'Raspberry Pi/MAGEEC energy measurement board cluster' discussed in Chapter 3. For such a setup to work, one would have to be careful as two devices cannot be considered alike due to subtle, and not so subtle, differences in operating system configurations and battery age. In proving that using genetic improvement to create approximate solutions in mobile computing devices is possible (albeit challenging), we hope the research may be carried out to make this a reality.

Chapter 5

Hardware specialisation and software parallelisation

Moving on from reducing energy consumption we decided to explore other properties of software which may be optimised. Parallelisation of sequential software is a problem many engineers find difficult despite the benefits of parallelisation being significant. Langdon and Harman used GI to enhance the parallelisation of Bowtie2, improving execution time 70x compared to the original [108]. nVidia have published case studies where utilisation of GPUs over traditional CPU, sequential computing can significantly reduce energy consumption and energy costs [135] — in one instance reducing a company's annual energy expenditure from $2.3 million (US) to $82,000 (US) by utilisation of GPU technology.

The benefits of parallelisation appear obvious. Suboptimal use of hardware affects both execution time and energy consumption, and, if rectified, can both improve without any degradation in functional correctness. Therefore, in this chapter we discuss research into utilising genetic improvement to automatically translate sequential software to parallel equivalents, with little human intervention, to reduce execution time.

5.1 Introduction

As the power of a single processing core reaches its limit, modern computer systems have become increasingly reliant on multicore architectures and accelerators; devices such as GPUs, FPGAs, and cryptography co-processors that are capable of improving the performance of specific computational tasks.

CUDA [131] and OpenCL [160] are languages that have been introduced to allow the development of software for GPUs. The VHSIC Hardware Description Language exists for FPGAs [118], and various frameworks and libraries exist to utilise multicore CPUs [54, 62]. However, utilising these languages, libraries, and frameworks is difficult for most developers as they do not lend themselves to traditional programming paradigms. In the domain of optimisation, software written to be executed sequentially cannot easily be translated to run on targets like multicore CPUs or GPUs, as these architectures can only increase performance via parallelisation. Learning to use these languages, frameworks, or libraries correctly is a large investment. Indeed, in many cases, utilisation

can considerably improve software performance but it is not economical due to the costs of human developers and the training they require.

In response to this, directive-based approaches have been developed. These allow engineers to create software in a manner in which they are comfortable and later add directives that inform the compiler to offload certain segments of code to other processing cores and/or hardware accelerators. Though simpler to use than CUDA, OpenCL, or other languages and frameworks, implementing these directives still requires training and incurs a cost in regards to developer time as the process of adding these directives to software is one of trial and error. Our goal in this research was to develop techniques to automatically insert these directives without any human intervention.

In this investigation we focused on optimising code to utilise the system's GPU. As such we studied automatically generating and inserting OpenACC [180] directives as OpenACC is, at the time of writing, the state-of-the-art when it comes to directive based GPGPU programming. It has been shown superior to its main rival, OpenMP [54] in this domain[1] [146].

To the best of our knowledge, there is only one other approach that automatically inserts OpenACC directives — a source-to-source compiler module called DawnCC [125]. DawnCC uses established static analysis techniques to determine where to insert OpenACC directives safely while incorporating range analysis to determine the range of arrays to copy from and to the system's GPU. However, we found that when DawnCC was run on the NAS-NPB benchmark suite (the suite of applications we later use to evaluate our approaches), DawnCC increased execution time in six of the seven applications (and could not influence execution time by a statistically significant extent in the remaining one)[2]. The reason for this is simply offloading code to the GPU does not guarantee faster code. For each parallelised region of code, there is an overhead where information must be transferred to the GPU for processing then transferred back upon completion. It is not at all uncommon for this transfer overhead to nullify any improvements that may be gained form parallelisation. This overhead is difficult, if not impossible, to determine at compile time.

So, while an important contribution, the techniques outlined in this paper differ from DawnCC in that they utilise genetic improvement [142]. In genetic improvement the problem faced by DawnCC is not so much a problem as the target software may be run and measured during optimisation using tests representative of usage during optimisation. Additionally, compilers must preserve the semantics of source code regardless of whether the semantics are efficient or necessary. Genetic improvement, on the other hand, only preserves the semantics specified by the user through the oracle (typically in an implicit manner via tests). This allows for more avenues of optimisation. We thought the application of genetic improvement to the automatic generation and insertion of OpenACC directives is a worthwhile endeavour and we were therefore the first to apply genetic improvement to parallelise code by the automatic generation and insertion of directives.

We investigated two genetic improvement approaches to automatically generate and insert OpenACC directives. One utilises grammar-based genetic programming [176] (GB-GP) to produce directives which are then inserted into the target source code with the execution time of the modified software (once compiled and run on a test input) used as the GB-GP's fitness measure. We refer to this as *GB-GP-Parallelisation*. The other takes a more bespoke approach. It inserts OpenACC directives to source code and tunes them in four separate steps, each of which uses either a greedy algorithm or an evolutionary strategy [19]. We refer to this as *Four-Stage-Parallelisation*.

We evaluated our approaches on a sequential variant of the NAS Parallel Benchmark suite [20]

[1] It should be noted, both OpenMP and OpenACC are similar standards so the techniques described in this chapter could be adapted to insert OpenMP directives if desired.

[2] This short investigation is outlined, in greater detail, in Appendix C.

provided by the Center for Manycore Programming at Seoul National University [50]. We refer to this collection of applications as the SNU-NPB suite. We ran both the GB-GP-Parallelisation and Four-Stage-Parallelisation approaches on each application within the SNU-NPB suite to produce variants that contained OpenACC directives. We compared the execution time of the sequential applications against these variants when running a test input to evaluate our approaches.

In order to make a truly fair evaluation as to the effectiveness of our approaches we needed a baseline truth; knowledge of what is achievable when attempting to optimise these sequential applications by inserting OpenACC directives. Fortunately we found such a baseline. Pino et al. parallelised the SNU-NPB suite using OpenACC directives as part of an investigation comparing OpenACC against OpenMP [146]. They have since made this hand-implemented OpenACC variant public [12]. We refer to this as the SNU-NPB-ACC suite. We used the performance of the SNU-NPB-ACC suite as a 'best-case scenario' proxy for our automatic OpenACC insertion techniques. With this setup, we asked the following research questions:

RQ1 How effective is GB-GP-Parallelisation?

- **RQ1a** What execution time reductions are achievable when using GB-GP-Parallelisation?
- **RQ1b** How do the reductions in execution time compare to handwritten OpenACC implementations?

RQ2 How effective is Four-Stage-Parallelisation?

- **RQ2a** What execution time reductions are achievable when using Four-Stage-Parallelisation?
- **RQ2b** How do the reductions in execution time compare to handwritten OpenACC implementations?

RQ3 What differs between the solutions produced by GB-GP-Parallelisation, Four-Stage-Parallelisation, and the handwritten OpenACC implementations?

5.2 Setup

In this section we discuss the design of both the GB-GP-Parallelisation and Four-Stage-Parallelisation techniques, the environment in which they were run, the applications in which they targeted, and the methodology we adhered by to obtain the data necessary to answer the research questions.

5.2.1 GB-GP-Parallelisation

Figure 5.1 shows the basic architecture of GB-GP-Parallelisation. At its core is a grammar-based genetic programming algorithm which produces a solution that is then translated into a patch that may be applied to the target source code to insert OpenACC directives. This patch is evaluated by a fitness function, with this fitness value returned to the grammar-based genetic programming algorithm. A grammar-based genetic programming algorithm functions like a traditional genetic programming (GP) algorithm [25] but has fixed rules (i.e. a grammar) which desribe how terminals and non-terminals may join in the tree. In a traditional grammar-based GP setup they are two inputs: the grammar, and a fitness function to evaluate the solutions it produces. The GB-GP-Parallelisation setup is more convoluted as it requires specific data from the target source code to

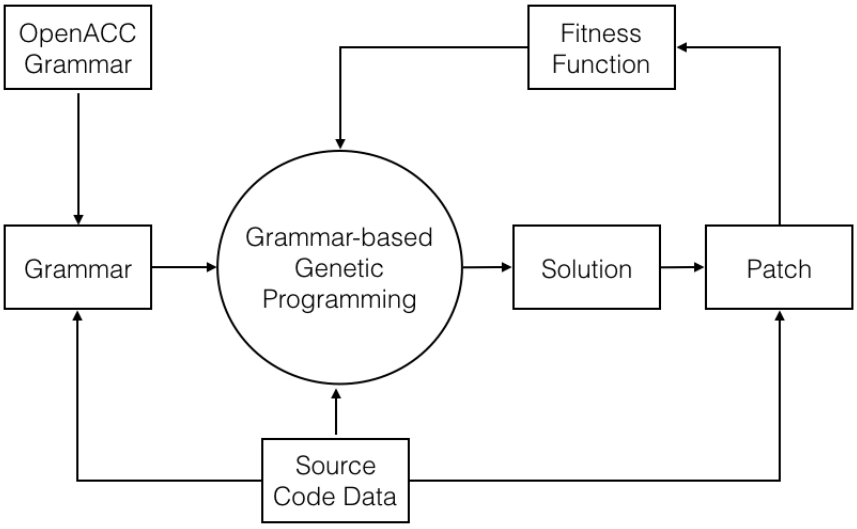


Figure 5.1: The architecture of GB-GP-Parallelisation.

```
<start>::= <start> | <start> | <directive>
<directive> ::= "#pragma acc parallel loop " <loop_line_number>
         | "#pragma acc wait " <line_number>
<loop_line_number> ::= "34@FileB.c" | "55@FileA.c"
<line_number> ::= "11@FileA.c" | "40@FileB.c" | "55@FileC.c"
```

Figure 5.2: The OpenACC grammar in Backus Normal Form (heavily abridged).

create both the grammar and to generate a patch from a solution output by the grammar-based GP.

Figure 5.2 shows an abridged snippet of the OpenACC grammar in Backus Normal Form (BNF), like that used in the GB-GP-Parallelisation setup. Grammar-based GP searches the set of solutions achievable within its grammar and, by design, is incapable of producing grammatically invalid solutions. As an example, for the grammar in Figure 5.2, `#pragma acc parallel loop 11@FileA` can never be generated as `11@FileA.c` is a production of `<line_number>`, not `<loop_line_number>`.

In practice the grammar is considerably more complex than this. The data necessary to produce a solution that will then be converted into a patch requires two components: what we call the 'OpenACC grammar' (which is a constant across all target source code, shown in Appendix A), and source code data. The OpenACC grammar is incomplete as, while it contains rules on how OpenACC directives must be structured for the compiler to interpret, it is missing production rules on where to insert these directives. GB-GP-Parallelisation appends these production rules to the OpenACC grammar during execution based on data extracted from the target source code. The full grammar (the OpenACC grammar plus the source code data) is thereby unique for each application. These appended production rules are the source code line numbers (`<line_number>`), the location of `FOR` loops (`<loop_line_number>`), the location of the first statement within a `FOR` loop (`<top_loop_line_number>`), and the location of functions (`<function_line_number>`). In our setup the grammar-based GP outputs solutions such as `#pragma acc parallel loop 12@FileA.c`. The postfix, `12@FileA.c`, is information unique to that application; a valid insertion point for the directive in this case. These solutions are then parsed into patches. The `#pragma acc parallel loop 12@FileA.c` example would result in a patch that inserts `#pragma acc parallel loop` at line `12` in `FileA.c`.

With the aforementioned production rules appended, the grammar-based GP can produce solutions, though translation is required to transform them into software patches. Some OpenACC directives declare how program variables are transferred to and from the GPU and main system memory. The patches therefore require the names of variables. In the grammar, variables are represented as integer placeholders, ranging from 1 to 100. These placeholders are translated into variable names when the solution is transformed into a patch. To translate an integer placeholder i into a variable we obtain a sorted vector V of all variables within the directive to be inserted's scope. We then select V_j from this vector where $j = i \mod |V|$. For example, the partial solution `#pragma acc parallel loop copyin(34)` with the vector of variables V equal to `[varA, varB, varC, varD]`, would select the $34 \mod |V|$th element in V (where 0 is the first element). In this case $34 \mod |V| = 2$, and thus `varC` would be chosen, resulting in the solution being translated to `#pragma acc parallel loop copyin(varC)`.

Finally there are circumstances when our grammar may introduce new program scopes (i.e., curly brackets in C/C++). We represent the end location of these scopes as a placeholder in the OpenACC grammar. Again, these range from 1 to 100. A solution produced by the grammar may be `#pragma acc kernels \n{35 15@FileB.c` which would be translated into a patch by inserting `#pragma acc kernels \n{` at line 15 in `FileB.c`, then inserting the closing bracket, `}`, 35 statements after this insertion, skipping inner scopes (such as `FOR`, `WHILE`, and `IF` statements). If the end of the current scope is reached, we start over from the directive insertion point.

To implement our grammar-based genetic programming approach, we used Epochx 1.4.1 [138], an open source genetic programming framework, written in Java, which supports grammar-based

genetic programming. As genetic programming algorithms can be setup in any number of ways, it is important to state how that contained in Epochx 1.4.1 functions.

Algorithm 8 The Genetic Programming Algorithm.

Require: G, the number of generations, $G \in \mathbb{N}$
C, the crossover rate, $C \in \mathbb{R} \wedge 0 \leq C \leq 1$
M, the mutation rate, $M \in \mathbb{R} \wedge 0 \leq M \leq 1$
S, the population size, $S \in \mathbb{N}$
T, the tournament size, $T \in \mathbb{N}$
initialise_population(S), returns an initial population of size S
evaluate(P), evaluates all the solutions in the population P
select(P, T), a tournament selection of size T on population P
crossover(p_1, p_2), crossover using parents p_1 and p_2
mutate(p), returns a mutant of solution p
get_random(), returns a random number $r : r \in \mathbb{R} \wedge 0 \leq r \leq 1$

1: $P \leftarrow$*initialise_population*(S)
2: *evaluate*(P)
3: **for** $1 \ldots G$ **do**
4: $P' \leftarrow \{\}$
5: **while** $|P'| < |P|$ **do**
6: $r \leftarrow$ *get_random*()
7: **if** $r < C$ **then**
8: $p_1 \leftarrow$ *select*(P, T)
9: $p_2 \leftarrow$ *select*(P, T)
10: $P' \leftarrow \{P'\} \cup \{$*crossover*$(p_1, p_2)\}$
11: **else if** $r < (C + M)$ **then**
12: $p \leftarrow$*select*(P)
13: $P' \leftarrow \{P'\} \cup \{$*mutate*$(p)\}$
14: **else**
15: $P' \leftarrow \{P'\} \cup \{$*select*$(P, T)\}$
16: **end if**
17: **end while**
18: $P \leftarrow P'$
19: *evaluate*(P)
20: **end for**
21: **return** x, where $x \in P \wedge x.fitness \neq -1 \wedge \forall p \in P : x.fitness \geq p.fitness$

Algorithm 8 outlines the genetic programming algorithm used in GB-GP-Parallelisation. Conforming to the grammar, a population P is initialised via the function *initialise_population*. The population is then evaluated via the *evaluate* method, giving each solution in the population a fitness value. The algorithm then iterates through a number of generations G. At the start of each new generation another population P' is derived from the previous generation P using crossover, mutation, and selection; the proportion of each based on the crossover rate C, and mutation rate M. Both *crossover* and *mutate* require solutions to be selected via *select*, a tournament selection of size T which selects based on the fitness of the solutions in the population P. Once the generation of P' is complete, P is replaced by P' and the population is evaluated. The best solution from the

Algorithm 9 Generating the Initial Population.

Require: S, the population size, $S \in \mathbb{N}$
$mutate(p)$, returns a mutant of solution p
$get\_initial\_solution()$, returns a single initial solution
$get\_random\_solution()$, returns a random solution

1: $P \leftarrow \{\}$
2: $i \leftarrow 0$
3: $N \leftarrow 3S$
4: **while** $|P| < S \wedge i \leq N$ **do**
5: $P \leftarrow \{P\} \cup \{get\_initial\_solution()\}$
6: $i \leftarrow i + 1$
7: **end while**
8: $i \leftarrow 0$
9: **while** $|P| < S \wedge i \leq N$ **do**
10: $p \leftarrow get\_initial\_solution()$
11: $P \leftarrow \{P\} \cup \{mutate(p)\}$
12: $i \leftarrow i + 1$
13: **end while**
14: **while** $|P| < S$ **do**
15: $P \leftarrow \{P\} \cup \{get\_random\_solution()\}$
16: **end while**
17: **return** P

final generation, based on the solutions' fitness, is then returned as the final, 'champion', solution.

The functionality of *initialise_population* is shown in Algorithm 9. The *initial_population* algorithm starts by attempting to populate set P with those generated by *get_initial_solution* which iterates through the target source code's `FOR` loops returning a solution which adds the `#pragma acc parallel loop` directive to this location. As it is possible that the number of `FOR` loops is less than the population size, and duplicates are not permitted, we limit the number of calls to *get_initial_solution* to three times the population size (N). If, after this, the population size is not met, *get_initial_solution* is called with its output mutated via the mutation operator. Again, this attempt is limited by three times the population size iterations (though given the search space of mutating one directive being high, this is rarely met in practise). Finally, if the population size is still not met, the rest of the initial population is filled with truly random solutions (via *get_random_solution*).

Our crossover operator takes two solutions as parents (selected using tournament selection) then produces a child with each directive, within each parent, having a 50% change of being included in the child solution. Figure 5.3 shows how our OpenACC grammar permits this type of crossover as directives (`<directive>`) may be taken away or added indefinitely.

Our mutation operator is based on the mutation operator first outlined by Whigham when he proposed grammar-based genetic programming [176]. Whigham mutation randomly selects a subtree and replaces it with a randomly generated (and grammatically correct) alternative. In our customisation, we do this 50% of the time. In the other 50% of the time we add a new directive to the solution, produced by calling *get_initial_solution*. This is our way of introducing (or reintroducing) sensible genetic material into the population.

```
<start>::= <start> <start> | <directive>
<directive>::= "#pragma acc " <choice>
<choice>::= "parallel "  <parallel> ...
| "loop " ... <loop_line_number>
| "parallel loop " ... <loop_line_number>
| "kernels " ... <line_number>
| "kernels loop " ... <loop_line_number>
| "data " ... <line_number>
...
```

Figure 5.3: The first three production rules in the OpenACC grammar (abridged).

When a solution produced by the grammar-based GP is to be evaluated, it is translated into an UNIX patch. However, before doing this some checks are run on the solution. There are some instances where the grammar produces incorrect or inefficient solutions that are easy to fix. The goal of this fix step is to reduce the number of ineffective evaluations. The first check ensures that no two directives share the same insertion point. If this is found, one directive is selected and another location (conforming to the grammar) within the source code is uniformly selected and the selected directive is updated to be inserted to this new location. This check iterates through all the directives within the solution until a pass is achieved where all directives have a unique insertion point. As it is feasible that all insertion locations are occupied, a maximum of five passes is allowed. If this maximum is met, the fixing is declared to have failed which automatically results in the solution being rejected, with a fitness value equal to that if it did not meet the fitness function hard constraints. Next there is a check to ensure that any instance of `#pragma acc loop` is contained within a `#pragma acc kernels loop`, `#pragma acc kernels`, or `#pragma acc parallel loop` construct. `#pragma acc loop` is inert when not present within these and, if such an instance is found, it is converted to `#pragma acc parallel loop` so it has an opportunity to optimise. Likewise, a `#pragma acc parallel loop` directive cannot be nested within a `#pragma acc parallel loop` or `#pragma acc kernels` construct. In such an instance the directive is changed to `#pragma acc loop`.

Once this 'fixing' step is complete the framework translates the solution into a UNIX patch. This is then passed to a script, bespoke to each application, which returns the fitness value. The contract of this script is it must return a positive double value, and this double tends higher for fitter solutions. A value of '-1' is returned if the solution breaks a hard constraint. In all the applications we targeted, we wrote the fitness function to return the reciprocal of execution time of the applications. If the application could not be compiled, produced an invalid output, or reached a timeout (10 minutes), we returned '-1'. In our setup this is translated to a fitness value equal to `java.lang.Double.MAX_VALUE` within the Epochx GB-GP evaluation procedure. When the script returns the fitness value, the corresponding solution in the GP algorithm is given that fitness value, and the GP algorithm continues.

5.2.2 Four-Stage-Parallelisation

Figure 5.4 shows the basic architecture of Four-Stage-Parallelisation. The optimisation happens over four distinct stages which we shall explain in this chapter. Optimisation starts with *insertion*

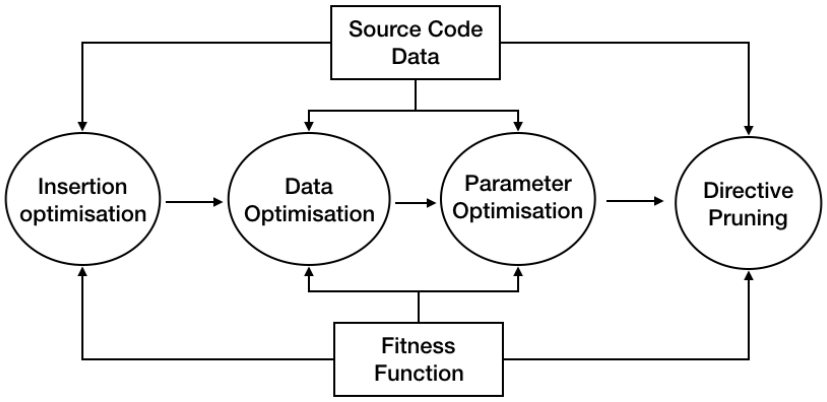


Figure 5.4: The Architecture of Four-Stage-Parallelisation

optimisation then moves through to *data optimisation*, *parameter optimisation*, and, finally, *directive pruning*. The origin of this design is that, unlike the process of GB-GP-Parallelisation, it more accurately represents the steps undertaken by a human implementing OpenACC directives. Adding a `#pragma acc parallel loop` directive to parallelise a `FOR` loop can increase execution time by a large amount until the correct parameters are added. Humans are aware of this and, as such, common advice on implementing OpenACC is to do so in stages, being aware that the product at the end of the earlier stages may be worse than the original, unmodified software. The different stages in the Four-Stage-Parallelisation take this consideration into account.

The first stage, *insertion optimisation*, is a greedy approach to annotating `FOR` loops as parallelisable (via the insertion of `#pragma acc parallel loop` directives). The goal in this stage is not to optimise for execution time but instead maximise the number of `FOR` loops that are parallelised. For each `FOR` loop within the program, a `#pragma acc parallel loop` directive is added, then the fitness function is run. In our setup a fitness function of '-1' indicates a failure to compile, a failure to preserve semantics, or a timeout. The *insertion optimisation* keeps a `#pragma acc parallel loop` insertions *if* the fitness function returns any value other than '-1'.

It should be noted that in the case of nested loops, an outer loop is always tested before its inner loops. As inserting a `#pragma acc parallel loop` directive cannot appear within a `FOR` loop already parallelised by a `#pragma acc parallel loop` directive (the compilation will fail), the framework has a bias to the parallelisation of outer loops. This is favourable as it results in more of the program being parallelised. When this stage is complete the solution with the most parallelised `FOR` loops (while preserving semantics and avoiding timeouts) is passed to the second stage.

The second stage, *data optimisation*, optimises the handling of variables that are initialised before a parallelised `FOR` loop structure but utilised within. In OpenACC each of these variables may be declared as either `copy`, `copyin`, `copyout`, `create`, or `present`. `copy` for when the variable must be copied into the GPU at the start of the parallelised `FOR` loop and copied out at the end. `copyin` for when the variable only needs to be copied into the GPU before computation but not copied back to main memory after. `copyout` for when the variable only needs to be copied back to main memory after (the variable is created on the GPU, but no value is moved from the system's main memory, it is assumed the value will be set within the GPU). `create` for when

the variable should be created within, and neither copied to or from, the GPU. Finally there is `present` which informs the compiler that the variable is already within the GPU and no action is needed. Without declaring a variable, the decision is left to the compiler. These declarations are appended to the end of OpenACC directives, for example: `#pragma acc parallel loop copy(variable)`.

To optimise this Four-Stage-Parallelisation represents each variable decision as a gene within a genotype. The value (or allele) of each gene is the corresponding variable's status within a specific parallelised `FOR` loop (we include a 'no status set' option to leave the decision to the compiler). A (1+1) evolutionary strategy (ES) [19] is carried out on this representation, with a mutation rate (the probability that any gene in the genotype is mutated while generating a variant) of $1/l$ where l is the number of genes/variables. A mutation operation on any gene will change its value to another state uniformly selected from all available states. Four-Stage-Parallelisation starts with an initial genotype with all its genes having a 'no status set' value. In this stage the fitness function is utilised in full (same fitness function as used for the GB-GP-Parallelisation). That is, the goal of the optimisation is to reduce execution time while preserving semantics. We take the best solution found during this stage and pass it forward to the next.

The *parameter optimisation* stage functions in a similar manner to *data optimisation* stage. For each `#pragma acc parallel loop` directive there are three parameters which are optimised — `num_gangs`, `num_workers`, and `vector_length`. In the OpenACC model of parallelisation there are three layers: 'gang', 'worker', and 'vector' The vector layer functions as a SIMD parallelisation; individual instructions working over multiple data elements. The `vector_length` is the number of data elements that may be operated on with the same instruction. The worker layer 'works' an individual vector. The `num_workers` specifies how many workers there are within a gang. A gang is a collection of workers that share a common cache memory. `num_gangs` specifies the number of gangs. There is no synchronisation or data communicated between gangs, each works completely independently. This model of parallelisation has been designed so OpenACC may target many different types of hardware — different types of GPUs as well as other hardware accelerators. Therefore, for each `FOR` loop parallelisation, for each hardware target, there is an optimal setting for the `num_gangs`, `num_workers`, and `vector_length` parameters. If these parameters are not specified they are determined by the compiler though they are seldom optimal.

In the Four-Stage-Parallelisation set, each of these parameters was set to 2^X where $X \in \mathbb{N} \wedge X \leq 10$. We set an upper-bound of 2^{10} as this seemed sensible given, in examples we found using these parameters in human-implemented OpenACC code, they rarely exceeded 2^8, and we found no examples of any parameters being set to 2^{10}. Like before, a special 'value not set' status was used to indicate the setting of a parameter was to be left to the compiler. As in the *data optimisation* stage, the space of parameter settings is represented as a series of genes. Every parallelised `FOR` loop has three genes within the genotype, one gene for each of the three parameters (`num_gangs`, `num_workers`, and `vector_length`). A (1+1)-ES is then used with a mutation rate of $1/l$ where l is number of genes/parameters to tune. The fitness function is the same as in the *data optimisation* stage; optimising the execution time while preserving program semantics. All the genes in the initial genotype are set to 'value not set' (i.e. the decision is left to the compiler). As in the previous stages, the best solution found is passed to the next, and final, stage.

The goal of the final stage, *directive pruning*, is to remove any directives which are not decreasing the programs execution time, even after optimising the handling of data and their parameters. This is done in a greedy manner. First the fitness of the current, full, solution, f_c was measured. Then, for each parallelised `FOR` loop, the `#pragma acc parallel loop...` directive is removed and

the fitness, f, measured. If $f \leq f_c$ then $f_c = f$. Otherwise, the directive is returned to the solution. It should be noted that *directive pruning* may return a solution where no loops are parallelised, or even a solution in which the execution time is greater than the original due to synergistic effects between OpenACC directives. This stage outputs the final solution.

In order to run either GB-GP-Parallelisation and Four-Stage-Parallelisation, a certain number of evaluations must be allocated. While in GB-GP-Parallelisation the number of evaluations can be set by altering the population size and number of generations, in Four-Stage-Parallelisation, the division of evaluations between stages must be carefully considered. To do so we first acknowledge that, for any application, the number of evaluations required by the *insertion optimisation* and *directive pruning* is constant. *Insertion optimisation* must evaluate all `FOR` loops, and *directive pruning* must evaluate all loops found to be parallelisable by the *insertion optimisation* step. Therefore, given a fixed evaluation budget, after running *insertion optimisation* we know the number of evaluations left to divide between the *data optimisation* and *parameter optimisation* stages. We first calculate what the maximum number of evaluations would be to exhaustively search each genotype in the *data optimisation* and *parameter optimisation* stages. We then weigh the *parameter optimisation*'s 'maximum number of evaluations' figure by 0.5. The reason for this is we observed that the *parameter optimisation* step is of low value compared to the *data optimisation* step. We then split the remaining number of evaluations (after taking into account that used/to be used by the *insertion optimisation* and *directive pruning* stages) between the *data optimisation* and *parameter optimisation* stages proportional to their respective, weighted, 'maximum number of evaluations' value.

5.2.3 Environment

We compiled the solutions produced by GB-GP-Parallelisation and Four-Stage-Parallelisation using the PGI 16.7 compiler. The PGI 16.7 compiler, provided by The Portland Group, compiles the OpenACC annotated source code under the OpenACC 2.5 standard. We ran the experiments on an Ubuntu 14.04.5 LTS Desktop system with an Intel Core i5-650 processor (3.2 GHz, 2 cores), 4GB of RAM and an nVidia GeForce GTX 1060 GPU. Genetic improvement implicitly optimises for the target hardware. Therefore any optimal solutions found in this investigation may not be optimal across all hardware targets [109]. We believe this to be an advantage of our approach rather than a hinderance. It allows software to be optimised in respect to the deployment environment. In some regards this work echoes previous work on using automatic parameter tuning to optimise kernels for specific GPU setups [134, 140]

5.2.4 Application Selection and profiling

In this work we targeted the Seoul National University NAS Parallel Benchmark (SNU-NPB) Suite [50], version 1.0.3. This benchmark suite is derived from the NAS Parallel Benchmark (NPB) suite [20], a collection of applications designed to benchmark parallel supercomputers. The NPB suite is written mostly in FORTRAN which neither approaches, at present, can process. The SNU-NPB variant has translated the suite to the C programming language and includes a sequential version of the applications, which we target in this investigation. The suite contains seven applications, the details of which are outlined in Table 5.1.

Application	Description	# C Files (Targeted)	LoC (Targeted)	Input Class	Execution Time (s)
BT	**B**lock **T**ridiagonal: Solves a synthetic system of non-linear partial differential equations using block tridiagonal matrices.	14 (6)	2,456 (1,419)	Custom*	14.39
CG	**C**onjugate **G**radiant: Using a conjugate gradient method, estimates the smallest eigenvalue of a large sparse symmetric positive-definite matrix.	2 (1)	491 (262)	Custom*	7.91
EP	**E**mbarrassingly **P**arallel: Generates independent Gaussian random variates using an 'embarrassingly parallel' approach.	4 (2)	365 (182)	A	25.92
FT	**F**ourier **T**ransform: Solves a three-dimensional partial differential equation using a fast Fourier transform.	6 (3)	613 (224)	A	7.93
LU	**L**ower-**U**pper: Solves a synthetic system of non-linear partial differential equations using a lower and upper triangular matrix.	18 (6)	2,183 (1,053)	W	5.48
MG	**M**ulti**G**rid: Estimates the solution of a 3-dimensional discrete Poisson equation using the V-cycle multi-grid method.	2 (2)	878 (387)	B	7.45
SP	**S**calar **P**entadiagonal: Solves a synthetic system of non-linear partial differential equations using a scaler pentadiagonal matrix.	17 (5)	1,944 (1,065)	W	5.23
TOTAL	—	63 (25)	8,930 (4,592)	—	74.31
AVERAGE	—	9.0 (3.6)	1,275.7 (656.0)	—	10.62

Table 5.1: The sequential applications within the NAS-NPB Suite. *Custom input classes defined in text.

The applications in the SNU-NPB suite each have a selection of input classes (input data with corresponding output data; essentially black box tests). When optimising the applications we use a single input class to train and evaluate on. For each application we selected the input class that ran for more than 5 seconds (so that smaller reductions in execution time could be detectable and not confused with statistical variance) but less than 30 seconds (to keep evaluation times at a manageable level). If two or more input classes fell within this range the one with the lowest execution time was chosen. If there were no input classes a custom input class was created.

Both BT and CG required custom classes to be created. A problem class for BT was created with 40x40x40 grids over 200 times steps with DT equal to 0.8×10^{-3}. For CG, we created a problem class with a size of 30,000 over 30 iterations.

For each application we only targeted a subset of the code for optimisation. To determine what subset to use we profiled each application using GPROF 2.24 (with the code compiled using GCC 4.8.4) running each application's respective input class. We focused on the level of C functions and selected the minimum set of functions that accounted for over 90% of execution time. We also included the applications' main methods as, in a manner of speaking, they account for 100% of execution time. The details of how many files/LOC were ultimately selected from this process is noted in Table 5.1 (in parenthesis).

Our reasoning for selecting the SNU-NPB suite above others is its applications are written in C (thereby compatible with our frameworks), they can be compiled by the PGI 16.7 compiler we use in our experiments, they contain test data so the correctness of program variants can quickly be verified, they run in a non-trivial amount of time (lower execution times make results more susceptible to statistical error), and we know that the applications are parallelisable. One other important factor in our decision is the existence of the SNU-NPB-ACC suite [12, 146] .

The SNU-NPB-ACC suite contains all the applications found within the SNU-NPB sequential suite but manually annotated with OpenACC directives, thereby parallelising them. We treat these hand-implemented OpenACC directives as a 'ground truth' for both GB-GP-Parallelisation and Four-Stage-Parallelisation — that is, a good estimate for what is achievable when targeting these applications. We compare the patches created by our approaches, for each application, against the solutions within the SNU-NPB-ACC suite in order to observe how our approaches may be modified to achieve better parallelisation; to behave more like a human expert.

Fitness Functions

Shown in Algorithm 10 is the template fitness function we use in our investigations. It should be noted, each application has its own fitness function with the application directory D, the execution timeout T, and the input data I hard-coded. This is because, in both our approaches, the fitness function is expected to accept only one parameter: the patch, P.

In this algorithm we allow the fitness function to evaluate the program without any optimisation. To do this no patch, P, is given. If present, the patch is applied then the application is compiled. If this compilation was unsuccessful, then '-1' is ultimately returned. If compiled successfully the application with its test input and program's output recorded O_r. We utilise the GNU project's `timeout` utility (version 8.21) to put an upper-limit on the evaluation time of a solution. The `timeout` utility produces a non-zero exit code for a timeout, otherwise return the exit code of the application. In Algorithm 10, we abstract this to a simple function which returns true if an exit code of zero was returned. If the application crashes during execution, the non-zero exit code is relayed through the timeout function (meaning the function *timeout* returns false in Algorithm 10). The

Algorithm 10 The skeleton fitness function used for evaluating each application

Require: D, the directory of the application
P, the input patch (optional)
T, execution timeout, $T \in \mathbb{N}$
I, input data
is_present(P), returns True if that P is present, otherwise false
apply_patch(P, D), returns the directory D with patch P applied
compile(D), compiles the code in D, returns True if successful
run(D, I), executes D on input I and returns the output
timeout(T, C), executes C, capping its execution to time T
is_successful(O), returns True of the output O , is valid
execution_time(O), returns the execution time from the output O

if *is_present*(P) **then**
  $D' \leftarrow$ *apply_patch*(P, D)
else
  $D' \leftarrow D$
end if
$R \leftarrow -1$
if *compile*(D') **then**
  if *timeout*$(T,$ $O_r \leftarrow$*run*$(D', I))$ **then**
    if *is_successful*(O_r) **then**
      $R \leftarrow$*execution_time*(O_r)
    end if
  end if
end if
return R

output of the program, O_r, contains two pieces of information: the execution time, and whether the application ran successfully (i.e. whether it produced the correct output for the corresponding input). Using O_r we output the execution time if the application run successfully and '-1' if it did not.

5.2.5 Methodology

We allocated 600 evaluations, for each application, when running both GB-GP-Parallelisation and Four-Stage-Parallelisation. For GB-GP-Parallelisation we set a population size of 60, over 10 generations, with a crossover and mutation rate of 0.33, and a tournament size of 6 (10%). The allocation of evaluations between the four stages for Four-Stage-Parallelisation is discussed in Section 5.2.2. For each we set the evaluation timeout (in the fitness function, see Algorithm 10) to 10 minutes.

When the two approaches completed execution, for each application, we ran the champion GB-GP-Parallelisation solution, the champion Four-Stage-Parallelisation solution, the original sequential application as found in the SNU-NPB, and the handwritten OpenACC implementation found in SNU-NPB-ACC, with the same input (see Table 5.1) 100 times, recording the execution time for each run.

With this data we then used the Wilcoxon rank sum test to declare whether GB-GP-Parallelisation or Four-Stage-Parallelisation were capable of reducing execution time by a statistically significant extent ($p < 0.05$) when compared the sequential source. We also noted the mean percentage change in execution time when comparing the solutions produced by GB-GP-Parallelisation and Four-Stage-Parallelisation compared to the sequential and handwritten OpenACC implementations for each application.

5.3 Results

In this section we show the results of carrying out the methodology outlined in Section 5.2.5, using the GB-SP-Parallelisation and Four-Stage-Parallelisation techniques described in Sections 5.2.1 and 5.2.2, to answer the research questions outlined in the chapter's introduction.

5.3.1 GB-GP-Parallelisation

To answer RQ1a, *What execution time reductions are achievable when using GB-GP-Parallelisation?*, we produced Figure 5.5. This bar chart shows the performance of GB-GP-Parallelisation; that champion variants' execution time is shown as a percentage of the original, unmodified, sequential equivalent. We found that GB-GP-Parallelisation reduces the execution time of CG, EP, and SP by 2.64%, 15.56%, 0.25%, on average, respectively. For all other applications no statistically significant change in execution time was observed. Treating those with no statistically significant change as a 0% reduction, we can say GB-GP-Parallelisation reduces execution time, across all applications, by 2.79% on average.

In answering RQ1b, *How do the reductions in execution time compare to handwritten OpenACC implementations?*, we produced Figure 5.6. This bar chart shows how these solutions compare to handwritten OpenACC implementations found in the SNU-NPB-ACC suite. As can be seen, only in one case does GB-GP-Parallelisation improve upon that produced by a human implementation.

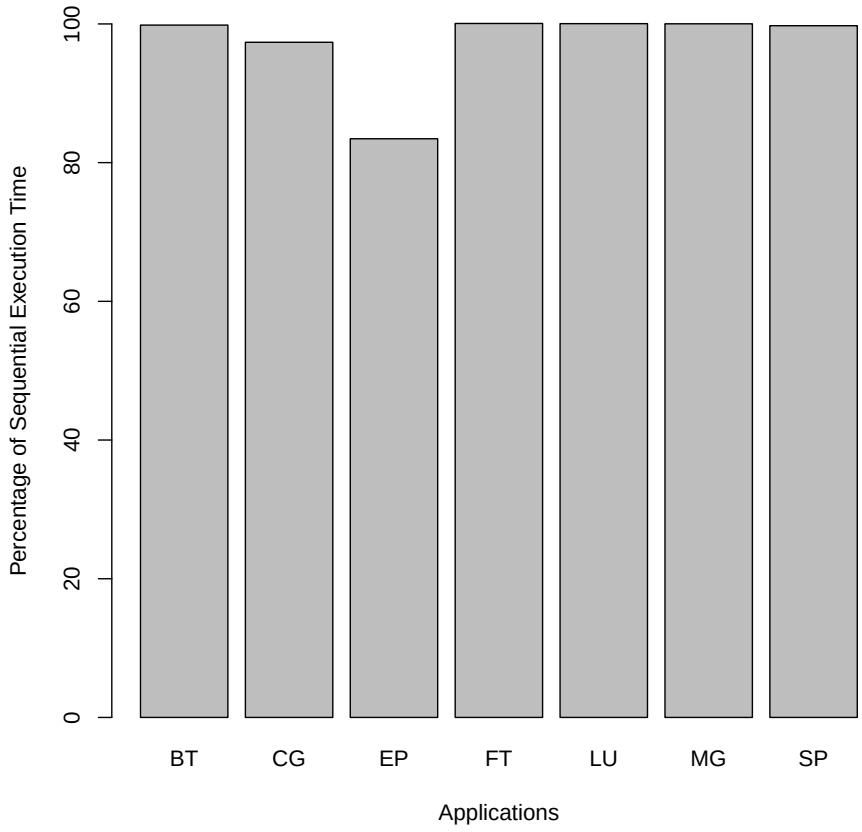


Figure 5.5: The performance of application optimisations using GB-GP-Parallelisation

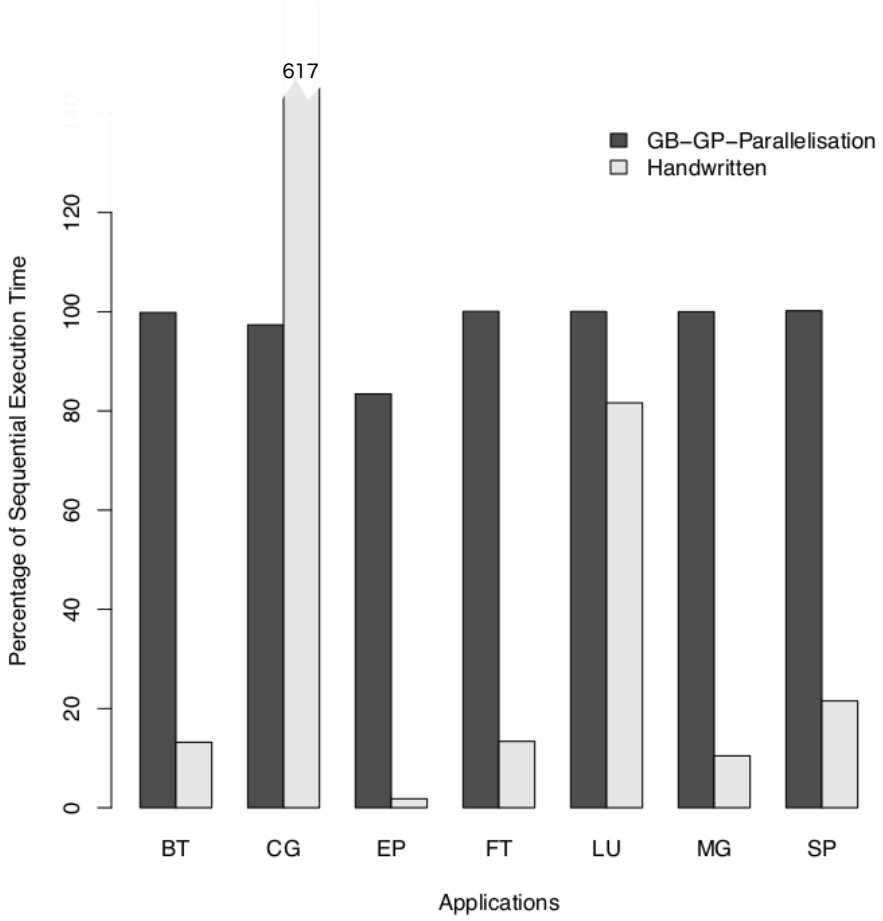


Figure 5.6: The performance of application optimisations using GB-GP-Parallelisation against the handwritten OpenACC implementation

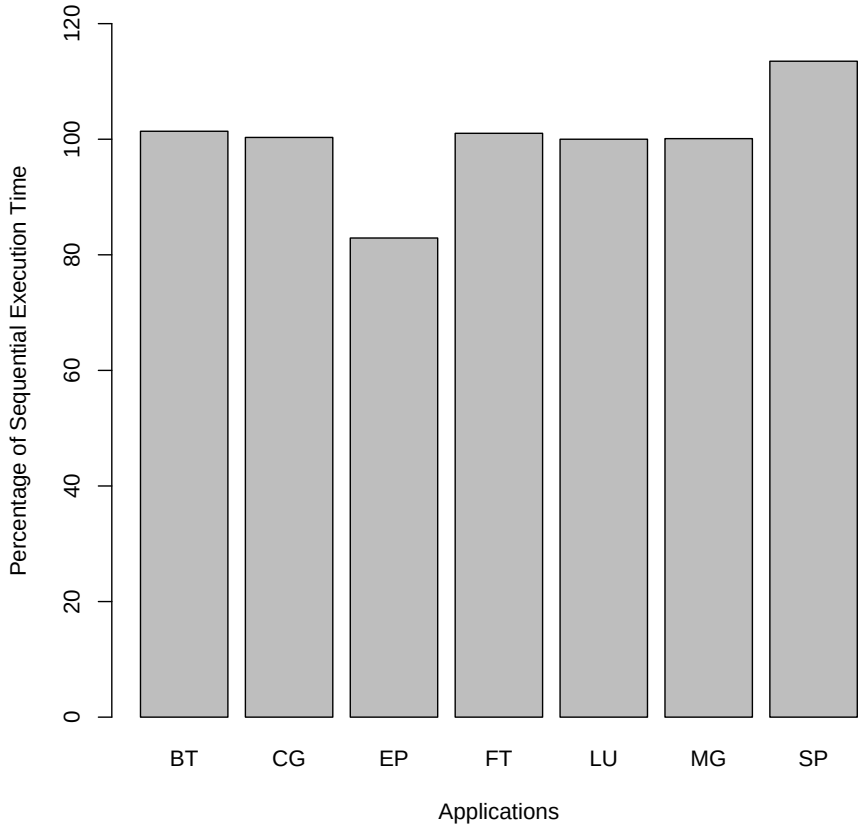


Figure 5.7: The performance of application optimisations using Four-Stage-Parallelisation

This one case is not due to the optimisation reducing execution time by a significant extent (the CG solution only reduces execution time by 2.64%). It is because the handwritten OpenACC implementation for CG increases execution time by 517%. We do not know why the handwritten CG implementation runs so poorly on our setup though we suspect this is due to some OpenACC optimisations simply not being universally beneficial across all hardware targets.

Treating CG as a 0% reduction, the handwritten OpenACC implementation reduces execution time by an average of 65.68%. It is therefore evident that GB-GP-Parallelisation could be improved upon significantly if we take the handwritten implementation as a guide to what is achievable.

5.3.2 Four-Stage-Parallelisation

In answering RQ2a, *What execution time reductions are achievable when using Four-Stage-Parallelisation?*, we produced the bar chart shown in Figure 5.7. Compared to that produced by GB-GP-Parallelisation, they are similar. A statistically significant decrease in execution time was only found in one case, EP, with a reduction of 17.09%. Due to how Four-Stage-Parallelisation works, it is possible for the chamption solution produced to increase execution time. This is due to the *directive pruning* stage, which utilises a greedy approach to selecting the optimal subset of loop parallelisations. In the case of LU, the chamption solution found contained no modifications to the source code (i.e. the 'directive pruning' stage removed all the inserted directives). In both Figure 5.7 and 5.8, we set LU as having an execution time of 100% of the sequential. Bar LU, all the effects found were statistically significant. In BT we observed an execution time increase of 1.37%, in CG a 0.31% increase, in FT a 1.02% increase, in MG a 0.10% increase, and in SP a 13.50% increase.

If we consider an increase in execution time to be a 0% decrease (as any user of Four-Stage-Parallelisation would choose the original sequential application in the case where it increased execution), we say Four-Stage-Parallelisation decreases execution time by 2.44% on average across all applications. This is slightly lower than GB-GP-Parallelisation's 2.79% reduction but the difference between the two is minimal. Given Four-Stage-Parallelisation's tendency to increase execution time, and the slightly lower reduction on average, GB-GP-Parallelisation appears superior to Four-Stage-Parallelisation. However, both are inferior when compared to the handwritten, OpenACC implementation.

In answering RQ2b, *How do the reductions in execution time compare to handwritten OpenACC implementations?*, we produced Figure 5.8. This bar chart shows how the optimisations produced by Four-Stage-Parallelisation compare against the handwritten OpenACC implementation. As in the case of GB-GP-Parallelisation, there is only one case in which a solution produced by Four-Stage-Parallelisation is superior, that of CG. Again this is due to the very poor performance of CG — the champion solution found by Four-Stage-Parallelisation for CG actually increases by 0.31%.

Just as in the case of GB-GP-Parallelisation, we can see there are significant performance gains that are not being exploited. The question therefore stands, what do the handwritten implementations do that GB-GP-Parallelisation and Four-Stage-Parallelisation do not?

5.3.3 Comparison to handwritten OpenACC

We compared the best solution for each application, for both the automatic parallelisation approaches, against the handwritten solutions in the SNU-NPB-ACC suite. In doing so we answered RQ3, *What differs between the solutions produced by GB-GP-Parallelisation, Four-Stage-Parallelisation, and the handwritten OpenACC implementations?* We attempted to explain what the handwritten solutions do which the solutions produced by the techniques outlined in this paper do not.

Upon comparing the solutions we found that the most significant difference is the SNU-NPB-ACC suite's use of OpenACC's `#pragma acc data` directive. These directives create what are known as 'data regions' — scopes of code for which data created or copied into the GPU will continue to exist until the scope is completed. For example, `#pragma acc data create(x){ [C code scope] }` will create the variable `x` on the GPU and it will exist on the GPU for the entirety of the `[C code scope]` segment. Both GB-GP-Parallelisation and Four-Stage-Parallelisation place emphasis on the insertion of `#pragma acc parallel loop` directives which create an implicit data region for the body of the `FOR` loop it parallelises. However, we found (when looking

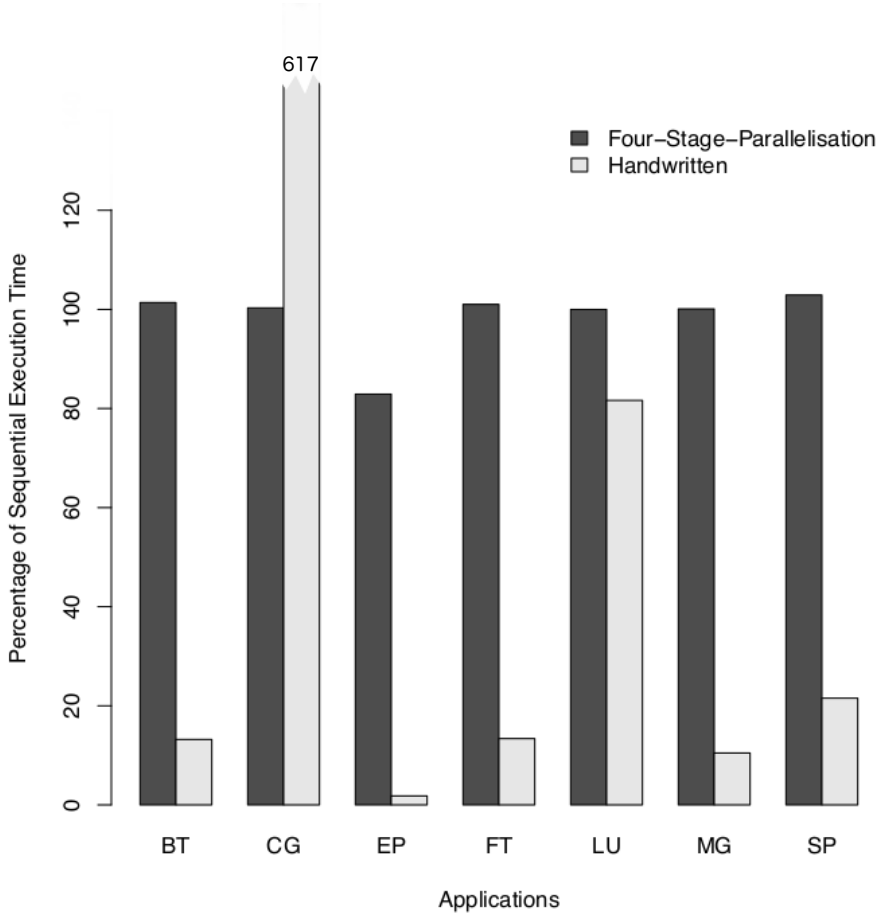


Figure 5.8: The performance of application optimisations using Four-Stage-Parallelisation against the handwritten OpenACC implementation

at the handwritten OpenACC implementations) that significant efficiencies can be gained when ensuring many parallelised loops exist in the same data region. We also found the SNU-NPB-ACC suite makes use of `acc_malloc`, a special function in OpenACC with similar functionality to the standard C library's `malloc` function except it allocates memory on the hardware accelerator (in this case the GPU) instead of main memory. This is another form of data optimisation.

In all the applications within the SNU-NPB-ACC suite we find that all the loop parallelisations are contained within a single data region. The role of this common data region is to create commonly utilised variables on the GPU at the beginning of the programs execution. Then, within this grand data region, the parallelised `FOR` loops declare that the data they need is already within the GPU and, therefore, it is not necessary to carry out the costly operation of copying from main computer memory or copying back when the loop has finished processing.

Figure 5.9 shows some (abridged) source code from the FT application in the SNU-NPB suite to demonstrate this. The `main` method contains a data directive which covers two methods, `init_ui` and `evolve`. This data directive 'creates' 11 arrays on the GPU (these are declared as global variables elsewhere in the program). Then, whenever a loop is parallelised and requires one of these variables, the `present` argument is used, as in `unit_ui` (line 20). This OpenACC directive informs the compiler that `u0_real`, `u0_imag`, `u1_real`, `u1_imag`, and `twiddle` are present on the GPU and, therefore, do not need to be copied over from main memory prior to processing or back to main memory after.

With such a setup it is important to copy the data back to main memory when it's needed and update the value in the GPU if changed in main memory (or vice-versa). OpenACC uses the terminology 'host' to refer to the systems main memory and 'device' to refer to the targeted hardware accelerator's memory, in this case, that of the GPU. This is the role of the `#pragma acc update host(<variable>)` and `#pragma acc update device(<variable>)` directives.

We find that, in the applications studied, many loops are parallelisable. However, simply adding a `#pragma acc parallel loop` directive is insufficient. For each `FOR` loop there is an overhead where data must be transferred to the GPU (device) memory before processing and back to the main (host) memory after. Frequently, this makes the parallelised version even more costly than the sequential. However, if these fixed overheads can be shared over multiple `FOR`-loop parallelisations the cost can be reduced. When the `#pragma acc data` directive is used effectively the transfer to and from the GPU can be kept at a minimum and it is this, we find, is when significant reductions in execution time can be achieved.

Both our approaches attempt to parallelise `FOR` loops, toggle their parameters, and optimise data flow solely within the `FOR` loop. It is possible in GB-GP-Parallelisation for a data directive to be created (it is contained within the grammar) but the search space is vast. Only in one instance, EP, did we find a data region being implemented, though this data region did not specify any variables and covered 3 statements which were not parallelised. This data region was, therefore, inert. It is clear comparing the solutions found via our approach, that we must encompass these `FOR` loop parallelisations into larger data regions.

We found that due to the lack of proper mechanisms to create meaningful data directives, neither approach parallelises many `FOR` loop structures. Table 5.2 shows the number of `FOR` loops parallelised using GB-GP-Parallelisation, and Four-Stage-Parallelisation champion solutions, compared against the handwritten implementations for each application. It should be noted that in cases of nested `FOR` loops, we only count the outer-most parallelisation. As can be seen, the handwritten implementation frequently parallelises more `FOR` loops than either GB-GP-Parallelisation and Four-Stage-Parallelisation approaches.

Application	GB-GP-Parallelisation	Four-Stage-Parallelisation	Handwritten
BT	0	1	44
CG	1	10	8
EP	3	2	5
FT	0	1	12
LU	0	0	59
MG	1	1	24
SP	0	2	65

Table 5.2: The number of `FOR` loop structures parallelised.

Application	Handwritten (s)	Sequential (s)	Handwritten without data directives (s)
BT	1.90	14.40	3463.23
CG	48.80	7.91	11.74
EP	0.47	25.92	48.11
FT	1.06	7.93	34.64
LU	4.47	5.23	970.11
MG	0.78	5.48	N/A (see text)
SP	1.24	5.48	1484.74

Table 5.3: The handwritten OpenACC implementation's execution time compared to its variant without data directives.

To emphasise what a dramatic effect correct use of data directives can have, we took the handwritten version and stripped it from OpenACC data directives (`#pragma acc data ...`), update statement (`#pragma acc update host/device`), data parameters on the `FOR` loop declarations (i.e. `copy`, `copyin`, `copyout`, `create`, and `present`), and replaced instances of `acc_malloc` with `malloc`. In cases where removal of these data directives resulted in a `FOR` loop parallelisation being uncompilable we manually removed the parallelisation. We recorded the execution time prior and after this change. These times can be seen in Table 5.3. Even compared to the execution time of the sequential time, parallelising loops without correctly applying data directives results in significantly increased execution time (in the case of MG, removing the data directives resulted in an error at execution time).

5.4 Discussion

Both GB-GP-Parallelisation and Four-Stage-Parallelisation employed different approaches to automatic optimisation. Both reduced execution time by less than 3%, 2.79% for GB-GP-Parallelisation and 2.44% for GB-GP-Parallelisation (averaged over all applications). We consider GB-GP-Parallelisation to be superior compared to Four-Stage-Parallelisation, not only in that it reduces execution time by a greater extent, but also that it does not produce any solutions that are worse than the original, which Four-Stage-Parallelisation does in five of the seven applications targeted. However, solutions that run in a higher execution time can easily be reverted back to the original, so GB-GP-Parallelisation's superiority is only slight.

In both instances the EP application shows the biggest reduction in execution time with GB-GP-Parallelisation achieving a 16.56% reduction and Four-Stage-Parallelisation achieving a 17.09% reduction. This is, perhaps, no surprise given this application's full name — 'embarrassingly parallel'. However, even in this case, we found we could not match the handwritten OpenACC implementation's performance. This handwritten OpenACC implementation is capable of reducing EP's execution time is 98.19%. The goal of this research was not to beat, or even match, that capable of a human expert (though such results would have been welcome). Rather we wished to develop a technique that could meaningfully parallelise software automatically. Just as a compiler will rarely produce the optimal machine code implementation for a given source, we do not envision a future in which genetic improvement is unbeatable, but one in which genetic improvement is 'good enough' — so cheap and easy to use that it can justify its existence on economic grounds. It is difficult to know at what point genetic improvement can begin to replace human effort, and will undoubtably differ on case-by-case basis. However, in this case, the optimisations produced by GB-GP-Parallelisation and Four-Stage-Parallelisation are lower than we would have hoped and below what can be achieved with a skilled human expert.

Our overarching idea in both the approaches to automatic parallelisation was to focus on parallelising `FOR` loops, optimising the data flow in and out of the loops, and tweaking their parameters. We have found this is too basic an idea and have concluded a more holistic one is required. We have found, when comparing to that produced by human experts, `FOR` loops share the same data and managing this data in a common way can significantly reduce the overheads of parallelisation. Moving data to and from the GPU is costly and this cost can (and in our experience, often does) destroy the gains of parallelisation. We believe future research should focus on this problem. It is one in which search-based techniques are well equipped. The objective is to optimise at what points data is transferred to and from main memory. This would not need to be a black-box optimisation as the source code reveals where data is used. For example, in the case that two parallelised `FOR` loops, one after another, use the same variables, it is logical that these `FOR` loops should share the same data region as data moved to the GPU for the first may as well continue to stay there for the second.

Once research has solved this problem we believe that we will begin to see significant execution time reductions for the applications studied. As is evident, the NAS Parallel benchmark suite studied consists of applications we know can be parallelised. We do not see this as a fault in our investigation. This is a new approach and therefore needs an easy example to try first. Just as we learn to walk before we learn to run, we target these benchmarks before moving into more challenging ones. In the medium term, once we have successfully optimised these benchmarks, we would like to test our techniques on 'real world' applications in which there exists no parallelised equivalent to see what can be achieved, and what other research considerations should be taken into account. If we can achieve noteworthy results in this domain then the software engineering community can begin to reap the significant benefits of automatic parallelisation with minimal cost.

```
1  int main(int argc, char *argv[]){
2  #pragma acc data create(u0_real, u0_imag,\\
3          u1_real, u1_imag, u_real, u_imag, twiddle,\\
4          gty1_real, gty1_imag, gty2_real, gty2_imag)
5    {
6      init_ui(dims[0], dims[1], dims[2]);
7      ...
8      for(iter = 1; iter <=niter; iter++){
9        evolve(dims[0], dims[1], dims[2]);
10       ...
11     }
12     ...
13   }
14   ...
15 }
16
17
18 static void init_ui(int d1, int d2, int d3){
19   int i, j, k;
20 #pragma acc parallel loop num_gangs(d3) num_workers(8)\\
21          vector_length(128) present(u0_real, u0_imag,\\
22          u1_real, u1_imag, twiddle)
23   for(k=0; k < d3; k++){
24     for(j=0; j < d2; j++){
25       for(i=0; i < d1; i++){
26         u0_real[k*d2*(d1+1) + j*(d1+1) + i] = 0.0;
27         u0_imag[k*d2*(d1+1) + j*(d1+1) + i] = 0.0;
28         u1_real[k*d2*(d1+1) + j*(d1+1) + i] = 0.0;
29         u1_imag[k*d2*(d1+1) + j*(d1+1) + i] = 0.0;
30         twiddle[k*d2*(d1+1) + j*(d1+1) + i] = 0.0;
31       }
32     }
33   }
34 }
35
36 static void evolve(int d1, int d2, int d3){
37   int i, j, k;
38 #pragma acc parallel loop present(u_real, u0_imag,\\
39          u1_real, u1_imag, twiddle)
40   for(k=0; k < d3; k++){
41     for(j=0; j < d2; j++){
42       u0_real[k*d2*(d1+1) + j*(d1+1) + i] = \\
43                 u0_real[k*d2*(d1+1) + j*(d1+1) + i] \\
44                 *twiddle[k*d2*(d1+1) + j*(d1+1) + i];
45       u0_imag[k*d2*(d1+1) + j*(d1+1) + i] = \\
46                 u0_imag[k*d2*(d1+1) + j*(d1+1) + i] \\
47                 *twiddle[k*d2*(d1+1) + j*(d1+1) + i];
48       u1_real[k*d2*(d1+1) + j*(d1+1) + i] = \\
49                 u0_real[k*d2*(d1+1) + j*(d1+1) + i];
50       u1_imag[k*d2*(d1+1) + j*(d1+1) + i] = \\
51                 u0_imag[k*d2*(d1+1) + j*(d1+1) + i];
52     }
53   }
54 }
```

Figure 5.9: Example usage of OpenACC data directives.

Chapter 6

Summary

In this thesis we have outlined research into the genetic improvement of non-functional properties. In this final chapter we shall outline the broad conclusions and take-away messages of this thesis, the current research building upon what has been described thus far, and future research directions we believe would be most advantageous to the field.

6.1 Contributions and Take-away Messages

Prior to carrying out this research, little was known about what the limits of genetic improvement of non-functional properties were. In this work we have tested Langdon and Harman's approach genetic improvement [108]. In particularly, we tested its applicability to energy optimisation. We found that the approach, which uses the *delete*, *copy*, and *replace* modification operators, is not ideal for this domain. We observed that, on average, only 2.69% improvement in energy efficiency was possible when preserving software semantics across four applications. Furthermore, it was shown that the vast majority of the effective operators were either *deletions* or *deletions-by-proxy* (i.e., *replace* operations that could be simplified via the application of a single *delete* operation). The relative ineffectiveness of Langdon and Harman's approach was further supported by our investigations into attempting to reduce 7zip and Bodytrack's energy consumption via GI, which we found to be a futile endeavour. However, we found evidence that interactions between operations produce synergistic and antagonistic interactions, thus supporting the belief that sophisticated search, such as evolutionary search, is required.

We also found that approximation of software output is a good avenue to improve the non-functional properties of software. In our investigation of the *delete*, *copy*, and *replace* operators, we observed that significantly larger reductions in energy consumption were possible when permitting a degradation in output quality. We explored this area further by using 'deep parameter optimisation' (a form of GI) to improve non-functional properties of software at the expense of correctness. In a study to reduce the execution time of a face detection algorithm, we were capable of achieving 48% decrease in runtime while reducing the classification accuracy of the face detection process from 98.96% to 98.2%. This, to us, verified what we had observed within our evaluation of the *delete*, *copy*, and *replace* operators — that permitting degradation in output can produce impressive improvements in non-functional properties of software.

Inspired by this, and due to having perviously focused on energy optimisation, we attempted

to use deep parameter optimisation to optimise the energy efficiency of an Android application running on a mobile device; a hardware target known to be energy constrained. Our investigations were hindered by the technical difficulties that, in highlighting and solving, we view as a valuable contribution to the genetic improvement research community in and of itself. We documented the problems we faced, from CPU throttling, through to noise produced by OS background processes. We also showed that optimisation of software on mobile devices is possible, though first these engineering challenges must be overcome.

We then moved on from this to discuss a different form of hardware optimisation — optimising software to run on GPUs. We created two different approaches to automatic parallelisation. Both involved the automatic generation and insertion of OpenACC directives, with one using a grammar-based genetic programming approach, and another which used a more bespoke, four-stage process, inspired by how human developers implement OpenACC directives. We found that our approaches did not parallelise software in any meaningful way compared by what was achievable by a human expert. Our conclusion from this research was that our approaches did not properly consider how data should be properly transferred to and from the GPU during computation; a factor in GPU utilisation which we found to have a big influence on the optimisation possible. For example, in one application (BT), we found that the hand-implemented optimisation, in which we had the data transfer optimisations removed, increased execution time from 14.40s to 3463.23s.

In summary, we would condense our contributions and take-away messages as follows:

- We have shown that the standard *delete*, *copy*, and *replace* operators, commonly used in GI research, produce a search space that is not easily navigable, and that the *replace* and *copy* operators are often ineffective or *deletions-by-proxy*.
- We have shown that synergy and antagonism exist within the GI search space and, therefore, search-based strategies such as evolutionary computation using crossover might be a useful way forward when navigating the space of possible code modification combinations.
- We have shown that approximation of output quality (even in modest amounts) can have a dramatic, positive effect on the improvement of non-functional properties.
- Our investigations into optimising the energy consumption of mobile devices has highlighted many engineering challenges which must be addressed in future research to allow for meaningful genetic improvement of energy consumption on these devices.
- Our investigation into automatic parallelisation of software has laid the first steps towards the automatic parallelisation of non-trivial sequential applications using genetic improvement. In doing so we have highlighted the issue of managing data between main system and GPU memory.

6.2 Current and Future Research

One might ask what future research to undertake, and is being undertaken, given the findings presented in this thesis. At present, work is being undertaken to incorporate the findings of Chapter 5 into a new automatic parallelisation framework that takes into account the data transfer problem we found. The new approach will utilise a constrained search to find the optimal transfer of data to and from GPU and main memory given a set parallelised `FOR` loops. It is hoped that this work

will prove genetic improvement's suitability to the task of automatic parallelisation and enable the field to aid developers in dealing with heterogeneous architectures.

Future work on energy consumption is needed. While in Chapter 4 we demonstrated, as a proof-of-concept, that Rebound, an Android application library, could be approximated to save energy consumption. We also highlighted the difficulty in getting reliable and consistent energy measurements. Given the importance of reducing energy consumption within these platforms, future research should focus on obtaining, what amounts to, an improved fitness function. I.e., we require better information on how modifications affect energy consumption. In our opinion, the community needs better energy measurement techniques, or energy estimation techniques, to evaluate an application. This, as we have observed, is challenging given the nature of complex, modern, computer systems.

We also showed in Chapter 3 that work is needed on the operators that are commonly used in GI. We found the standard *delete*, *copy*, and *replace*, operators are largely ineffective (though they are more effective when output approximation is permitted). This should motivate future research into investigating more clever operators. The *delete*, *copy*, and *replace* operators work at the level of source-code lines, whereas operators which are capable of manipulating multiple lines may function better. Work, for example, on design patterns has shown significant energy savings are possible when refactoring them in specific ways [133]. As the refactoring of these requires the modification of many lines, usually over many files, operators which modify single lines are unlikely to find such improvements. Research is required to see how such refactoring could be formulated as GI operators to then be applied to source code. We have, however, given evidence that search approaches such as those found in evolutionary computation are likely to be more effective than more exploitative approaches due to the presence of antagonism in the search space. We have therefore given future genetic improvement framework developers advice: use search-based techniques that utilise exploration as the energy space space contains local optima.

6.3 Final Remarks

In this thesis we have explored, and attempted to expand, the field of genetic improvement of non-functional properties. The investigations outlined in this thesis answer questions, but also, in doing so, ask more. We have shown optimisation of energy consumption is possible, but that questions remain on the best way to do so. We have shown that approximation may be used to effectively improve software's non-functional properties, to a much greater extent that when approximation is not permitted. However, we've also shown the need for more research into how we can effectively measure these non-functional properties, particularly in new hardware targets such as mobile devices and their energy-constrained nature. As, if we do not, optimisation remains difficult, whether or not approximation is permitted. We have shown early work into utilising genetic improvement to automatically translate software to modern, parallel hardware targets. Though we have shown this task is harder than we had initially anticipated, as we are among the first to do so, we have outlined new problems that require solutions.

In our introduction we outlined our vision of genetic improvement — the possibility of automation of non-functional property improvement. Though, such an ambitious goal will likely take several decades to accomplish, it is hoped that the research outlined here will aid future researchers, by providing answers to their questions and inspiring them with questions that have yet to be answered.

Bibliography

[1] 7zip. http://www.7-zip.org. Accessed: 12-April-2017.

[2] Cobertura : A code coverage utility for Java. http://cobertura.github.io/cobertura. Accessed: 23-March-2017.

[3] DawnCC — A source-to-source compiler for parallelizing C/C++ programs with code annotations. https://github.com/gleisonsdm/DawnCC-Compiler. Accessed: 7-November-2017.

[4] Evosuite. http://www.evosuite.org/. Accessed: 12-April-2017.

[5] MAGEEC energy measurement board. http://mageec.org/wiki/Power_Measurement_Board. Accessed: 12-April-2017.

[6] MAX17047/MAX17050 ModelGauge m3 Fuel Gauge. Accessed: 01-November-2016.

[7] MOEA Framework: A free and open source Java framework for multiobjective optimization. http://www.moeaframework.org. Accessed: 23-March-2017.

[8] Omxplayer. https://github.com/popcornmix/omxplayer. Accessed: 12-April-2017.

[9] Raspberry Pi. https://www.raspberrypi.org. Accessed: 12-April-2017.

[10] Raspbian. http://www.raspbian.org. Accessed: 12-April-2017.

[11] Rebound: Spring animations for Android. http://facebook.github.io/rebound. Accessed: 23-March-2017.

[12] NAS_OpenACC_2.5. https://github.com/spino327/NAS_OpenACC_2.5, 2017. Accessed: 3-August-2017.

[13] Michael Affenzeller, Stefan Wagner, Stephan Winkler, and Andreas Beham. *Genetic algorithms and genetic programming: modern concepts and practical applications*. CRC Press, 2009.

[14] Android. *Android Power Profiles*. Accessed: 01-March-2016.

[15] Android Developers. Optimizing for doze and app standby. https://developer.android.com/training/monitoring-device-state/doze-standby.html. Accessed: 23-March-2017.

[16] Sebastian Anthony. Facebook's evolutionary search for crashing software bugs. https://arstechnica.com/information-technology/2017/08/facebook-dynamic-analysis-software-sapienz, 2017. Accessed: 12-April-2018.

[17] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[18] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the 2008 Congress on Evolutionary Computation — CEC '08*. IEEE, 2008.

[19] Thomas Back, Frank Hoffmeister, and Hans-Paul Schwefel. A survey of evolutionary strategies. In *Proceedings of the 1991 International Conference on Genetic Algorithms*, volume 2. Morgan Kaufmann, 1991.

[20] David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Robert L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Thomas A Lasinski, Rob S. Schreiber, et al. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[21] Mutsunori Banbara, Haruki Matsunaka, Naoyuki Tamura, and Katsumi Inoue. Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. In *Proceedings of the 2010 International Conference on Logic for Programming Artificial Intelligence and Reasoning — LPAR '10*, pages 112–126. Springer, 2010.

[22] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 2014 Symposium on the Foundations of Software Engineering — FSE '14*, pages 588–598, 2014.

[23] Abhijeet Banerjee and Abhik Roychoudhury. Automated re-factoring of Android apps to enhance energy-efficiency. In *Proceedings of the Conference on Mobile Software Engineering and Systems — MOBILESoft '16*, pages 139–150. IEEE, 2016.

[24] Wolfgang Banzhaf and William B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3(1):81–91, 2002.

[25] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming: An introduction*. Morgan Kaufmann, 1998.

[26] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 2014 Symposium on the Foundations of Software Engineering — FSE '14*, pages 306–317. ACM, 2014.

[27] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis — ISSTA '15*, pages 257–269. ACM, 2015.

[28] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[29] Morris C. Berenbaum. What is synergy? *Pharmacological Reviews*, 41(2):93–141, 1989.

[30] James C. Bezdek, Srinivas Boggavarapu, Lawrence O. Hall, and Amine Bensaid. Genetic algorithm guided clustering. In *Proceedings of the 1st Conference on Evolutionary Computation*, pages 34–39. IEEE, 1994.

[31] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[32] John A. Birdsell and Christopher Wills. The evolutionary origin and maintenance of sexual recombination: a review of contemporary models. In *Evolutionary Biology*, pages 27–138. Springer, 2003.

[33] Mahmoud Bokhari and Markus Wagner. Optimising energy consumption heuristically on Android mobile phones. In *Proceedings of the 2016 Genetic and Evolutionary Computation Conference Companion — GECCO Companion '16*, pages 1139–1140. ACM, 2016.

[34] Mahmoud A. Bokhari, Bobby R. Bruce, Brad Alexander, and Markus Wagner. Deep parameter optimisation on Android smartphones for energy minimisation — A tale of woe and proof-of-concept. In *Proceedings of the 2017 Genetic and Evolutionary Computation Conference Companion — GECCO Companion '17*. ACM, 2017.

[35] Mahmoud A Bokhari, Yuanzhong Xia, Bo Zhou, Brad Alexander, and Markus Wagner. Validation of internal meters of mobile Android devices. *arXiv preprint arXiv:1701.07095*, 2017.

[36] Boston Consulting Group. GeSI SMARTer2020: The role of ICT in driving a sustainable future. `http://gesi.org/SMARTer2020`, 2012. Accessed: 12-April-2017.

[37] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In *Proceedings of the 2005 International Conference on Concurrency Theory — CONCUR '05*, pages 488–502. Springer, 2005.

[38] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008.

[39] Alexander Edward Ian Brownlee, Nathan Burles, and Jerry Swan. Search-based energy optimization of some ubiquitous algorithms. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):188–201, 2017.

[40] Bobby R. Bruce. Deep parameter optimisation for face detection using the Viola-Jones algorithm in OpenCV: A correction. Technical Report RN/17/07, Department of Computer Science, University College London, 2017.

[41] Bobby R. Bruce, Jonathan M. Aitken, and Justyna Petke. Deep parameter optimisation for face detection using the Viola-Jones algorithm in OpenCV. In *Proceedings of the 2016 Symposium on Search-Based Software Engineering — SSBSE '16*, pages 238–243. Springer, 2016.

[42] Bobby R. Bruce and Justyna Petke. Towards automatic generation and insertion of OpenACC directives. Technical Report RN/18/04, Department of Computer Science, University College London, 2018.

[43] Bobby R. Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Genetic and Evolutionary Computation Conference — GECCO '15*. ACM, 2015.

[44] Bobby R. Bruce, Justyna Petke, Mark Harman, and Earl T. Barr. Approximate oracles and synergy in software energy search spaces. *IEEE Transactions on Software Engineering*, 2018. (To appear).

[45] Bruce Buchanan, Georgia Sutherland, and Edward A. Feigenbaum. *Heuristic DENDRAL: A program for generating explanatory hypotheses in organic chemistry*. Defence Technical Information Center, 1968.

[46] Christian Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour. Choosing the "best" sorting algorithm for optimal energy consumption. In *Proceedings of the 2009 International Conference on Software Technologies — ICSOFT '09*, pages 199–206. Springer, 2009.

[47] Nathan Burles, Edward Bowles, Alexander EI Brownlee, Zoltan A Kocsis, Jerry Swan, and Nadarajen Veerapen. Object-oriented genetic improvement for improved energy consumption in Google Guava. In *Proceedings of the 2015 Symposium on Search Based Software Engineering — SSBSE '15*, pages 255–261. Springer, 2015.

[48] Nathan Burles, Edward Bowles, Bobby R. Bruce, and Komsan Srivisut. Specialising Guava's cache to reduce energy consumption. In *Proceedings of the 2015 Symposium on Search-Based Software Engineering — SSBSE '15*, pages 276–281. Springer, 2015.

[49] Canalys. Smart phones overtake client PCs in 2011. `https://www.canalys.com/static/press_release/2012/canalys-press-release-030212-smart-phones-overtake-client-pcs-2011_0.pdf`, 2012. Accessed: 24-March-2017.

[50] Center for Manycore Programming. SNU NPB suite. `http://aces.snu.ac.kr/software/snu-npb`, 2013. Accessed: 26-June-2017.

[51] Pinhong Chen and Kurt Keutzer. Towards true crosstalk noise analysis. In *Proceedings of the 1999 International Conference on Computer-Aided Design — ICCAD '99*, pages 132–138. IEEE, 1999.

[52] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 2013 Design Automation Conference — DAC '13*. ACM, 2013.

[53] Michael Codish, Igor Gonopolskiy, Amir M. Ben-Amram, Carsten Fuhs, and Jürgen Giesl. SAT-based termination analysis using monotonicity constraints over the integers. *Theory and Practice of Logic Programming*, 11(4-5):503–520, 2011.

[54] Leonardo Dagum and Ramesh Menon. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational science and engineering*, 5(1):46–55, 1998.

[55] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. John Murray, 1859.

[56] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory power estimation and capping. In *Proceedings of the 2010 International Symposium on Low-Power Electronics and Design — ISLPED '10*, pages 189–194. IEEE, 2010.

[57] Kenneth A De Jong. On using Genetic Algorithms to search program spaces. In *Proceedings of the 1997 International Conference on Genetic Algorithms — ICGA '87*, pages 210–216, 1987.

[58] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. A. M. T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[59] Nachum Dershowitz, Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1):117–156, 2001.

[60] Christos Dimopoulos and Neil Mort. A hierarchical clustering methodology based on genetic programming for the solution of simple cell-formation problems. *International Journal of Production Research*, 39(1):1–19, 2001.

[61] Laurence Charles Ward Dixon and Georges Philip Szegö. *Towards global optimisation*. North-Holland Amsterdam, 1978.

[62] Alastair Donaldson, Colin Riley, Anton Lokhmotov, and Andrew Cook. Auto-parallelisation of Sieve C++ programs. In *Proceedings of the 2007 European Conference on Parallel Processing — Euro-Par '07*, pages 18–27. Springer, 2007.

[63] Mian Dong and Lin Zhong. Chameleon: a color-adaptive web browser for mobile OLED displays. *IEEE Transactions on Mobile Computing*, 11(5):724–738, 2012.

[64] Mian Dong and Lin Zhong. Power modeling and optimization for oled displays. *IEEE Transactions on Mobile Computing*, 11(9):1587–1599, 2012.

[65] Pedro G. Espejo, Sebastian Ventura, and Francisco Herrera. A survey on the application of Genetic Programming to classification. *IEEE Transactions on Systems, Man, and Cybernetics — Part C: Applications and Reviews*, 40(2):121–144, 2010.

[66] Edward A. Feigenbaum and Bruce G. Buchanan. DENDRAL and META-DENDRAL: Roots of knowledge systems and expert system applications. *Artificial Intelligence*, 59(1):233–240, 1993.

[67] Ronald Aylmer Fisher. *The genetical theory of natural selection*. Dover Publications, 1958.

[68] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A Genetic Programming approach to automated software repair. In *Proceedings of the 2015 Genetic and Evolutionary Computation Conference — GECCO '09*, pages 947–954. ACM, 2009.

[69] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 2011 Symposium on the Foundations of Software Engineering — FSE '11*, pages 416–419. ACM, 2011.

[70] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the 2010 International Symposium on the Foundations of Software Engineering — FSE '10*, pages 147–156. ACM, 2010.

[71] Jürgen Giesl. Termination analysis for functional programs using term orderings. *Static Analysis*, pages 154–171, 1995.

[72] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, et al. Proving termination of programs automatically with AProVE. In *Proceedings of the 2014 International Joint Conference on Automated Reasoning — IJCAR '14*, pages 184–191. Springer, 2014.

[73] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings of the 2006 International Joint Conference on Automated Reasoning — IJCAR '06*, pages 281–286. Springer, 2006.

[74] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[75] Fred Glover. Tabu search — part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[76] Fred Glover. Tabu search — part ii. *ORSA Journal on computing*, 2(1):4–32, 1990.

[77] David E. Goldberg and John H. Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[78] Michael A. Goodrich, Bryan S. Morse, Damon Gerhardt, Joseph L Cooper, Morgan Quigley, Julie A. Adams, and Curtis Humphrey. Supporting wilderness search and rescue using a camera-equipped mini UAV. *Journal of Field Robotics*, 25(1-2):89–110, 2008.

[79] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[80] Gregory Griffin, Alex Holub, and Pietro Perona. Caltech-256 object category dataset. 2007.

[81] Jie Han and Michael Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *Proceedings of the 2013 European Test Symposium — ETS '13*, pages 1–6. IEEE, 2013.

[82] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering — ICSE '13*, pages 92–101. IEEE, 2013.

[83] Saemundur Oskar Haraldsson, John R. Woodward, Alexander E. I. Brownlee, and Kristin Siggeirsdottir. Fixing bugs in your sleep: How genetic improvement became an overnight success. In *Proceedings of the 2017 Genetic and Evolutionary Computation Companion — GECCO Companion '12*. ACM, 2012.

[84] Mark Harman, Yue Jia, and William B Langdon. Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In *Proceedings of the 2014 Symposium on Search Based Software Engineering — SBSE '14*, pages 247–252. Springer, 2014.

[85] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[86] Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea Arcuri, and John A. Clark. The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs. In *Proceedings of the 2012 International Conference on Automated Software Engineering — ASE '12*, pages 1–14. IEEE, 2012.

[87] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11, 2012.

[88] Mark Harman and Phil McMinn. A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis — ISSTA '07*, pages 73–83. ACM, 2007.

[89] John Heggestuen. Business Insider: One In Every 5 People IN The World Own A Smartphone, One in Every 17 Own A Tablet. `http://www.businessinsider.com/smartphone-and-tablet-penetration-2013-10`, 2013. Accessed: 12-April-2017.

[90] Timothy Hickey, Qun Ju, and Maarten H. Van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.

[91] Robert Hinterding, Harry Gielewski, and Thomas C. Peachey. The nature of mutation in genetic algorithms. In *Proceedings of the 1995 International Conference on Genetic Algorothms — ICGA '95*, pages 65–72, 1995.

[92] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *SIGPLAN Notices*, volume 46, pages 199–212. ACM, 2011.

[93] John H. Holland. *Adaptation in natural and artificial systems.* MIT Press, 1975.

[94] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, 2007.

[95] Matti Järvisalo, Petteri Kaski, Mikko Koivisto, and Janne H Korhonen. Finding efficient circuits for ensemble computation. In *Proceedings of the 2012 International Conference on Theory and Applications of Satisfiability Testing — SAT '12*, pages 369–382. Springer, 2012.

[96] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[97] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. How easy is local search? *Journal of computer and system sciences*, 37(1):79–100, 1988.

[98] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Benchmark precision and random initial state. In *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems — SPECTS '05*, pages 484–490. IEEE, 2005.

[99] A. H. G. Rinnooy Kan and G. T. Timmer. Stochastic methods for global optimization. *American Journal of Mathematical and Management Sciences*, 4(1-2):7–40, 1984.

[100] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 1992 European Conference on Artificial Intelligence — ECAI '92*, pages 359–363. Springer, 1992.

[101] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015.

[102] Jonathan G. Koomey. Worldwide electricity used in data centers. *Environmental Research Letters*, 3(3), 2008.

[103] Christos Koulamas, SR Antony, and R. Jaen. A survey of simulated annealing applications to operations research problems. *Omega*, 22(1):41–56, 1994.

[104] John R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

[105] Axel van Lamsweerde. Formal specification: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159. ACM, 2000.

[106] Jason Landsborough, Stephen Harding, and Sunny Fugate. Removing the kitchen sink from software. In *Proceedings of the 2015 Genetic and Evolutionary Computation Conference — GECCO '15*, pages 833–838. ACM, 2015.

[107] William B. Langdon and Mark Harman. Evolving a CUDA kernel from an nVidia template. In *Proceedings of the 2010 Congress on Evolutionary Computation — CEC '10*, pages 1–8. IEEE, 2010.

[108] William B. Langdon and Mark Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, 2015.

[109] William B. Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3D medical image registration CUDA software with genetic programming. In *Proceedings of the 2014 Genetic and Evolutionary Computation Conference — GECCO '14*, pages 951–958. ACM, 2014.

[110] William B. Langdon, Justyna Petke, and Bobby R. Bruce. Optimising quantisation noise in energy measurement. In *Proceedings of the Conference on Parallel Problem Solving from Nature — PPSN '16*, pages 249–259. Springer, 2016.

[111] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 2012 International Conference on Software Engineering — ICSE '12*, pages 3–13. IEEE, 2012.

[112] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[113] Claire Le Goues, Westley Weimer, and Stephanie Forrest. Representations and operators for improving evolutionary software repair. In *Proceedings of the 2012 Genetic and Evolutionary Computation Conference — GECCO '12*, pages 959–966. ACM, 2012.

[114] Ding Li, Shuai Hao, Jiaping Gui, and William G. J. Halfond. An empirical study of the energy consumption of Android applications. In *Proceedings of the 2014 International Conference on Software Maintenance and Evolution — ICSME '14*, pages 121–130. IEEE, 2014.

[115] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. Calculating source line level energy information for Android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis — ISSTA '13*, pages 78–89. ACM, 2013.

[116] Ding Li, Angelica Huyen Tran, and William G. J. Halfond. Making web applications more energy efficient for OLED smartphones. In *Proceedings of the 2014 International Conference on Software Engineering — ICSE '14*, pages 527–538. ACM, 2014.

[117] Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of logic systems. In *Proceedings of the 1997 International Conference on Logic Programming — ICLP '97*, page 63. MIT Press, 1997.

[118] Roger Lipsett, Carl F. Schaefer, and Cary Ussery. *VHDL: Hardware description and design*. Springer Science & Business Media, 2012.

[119] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[120] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering — ICSE '16*, pages 237–248. ACM, 2016.

[121] Irene Manotas, Lori Pollock, and James Clause. SEEDS: a software engineer's energy-optimization decision support framework. In *Proceedings of the 2014 International Conference on Software Engineering — ICSE '14*, pages 503–514. ACM, 2014.

[122] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis — ISSTA '16*, pages 94–105. ACM, 2016.

[123] Alexandru Marginean, Earl T. Barr, Mark Harman, and Yue Jia. Automated transplantation of call graph and layout features into kate. In *Proceedings of the 2015 Symposium on Search Based Software Engineering — SSBSE '15*, pages 262–268. Springer, 2015.

[124] R. Timothy Marler and Jasbir S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

[125] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. DawnCC: automatic annotation for data parallelism and offloading. *ACM Transactions on Architecture and Code Optimization*, 14(2):13, 2017.

[126] Melanie Mitchell, Stephanie Forrest, and John H. Holland. The Royal Road for Genetic Algorithms: Fitness landscapes and GA performance. In *Proceedings of the 1992 European Conference on Artificial Life — ECAL '92*, pages 245–254, 1992.

[127] Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys*, 51(1):17, 2018.

[128] Vojtech Mrazek, Zdenek Vasicek, and Lukas Sekanina. Evolutionary approximation of software for embedded systems: Median function. In *Proceedings of the 2015 Genetic and Evolutionary Computation Conference Companion — GECCO Companion '15*, pages 795–801. ACM, 2015.

[129] Hermann Joseph Muller. Some genetic aspects of sex. *The American Naturalist*, 66(703):118–138, 1932.

[130] Allen Newell, John C. Shaw, and Herbert A. Simon. Report on a general problem-solving program. In *IFIP Congress*, pages 256–264, 1959.

[131] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[132] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11, 2011.

[133] Adel Noureddine and Ajitha Rajan. Optimising energy consumption of design patterns. In *Proceedings of the 2015 International Conference on Software Engineering — ICSE '15*, pages 623–626. IEEE, 2015.

[134] Cedric Nugteren. CLBlast: A tuned OpenCL BLAS library. *arXiv preprint arXiv:1705.05249*, 2017.

[135] nVidia Corporation. Doing more with less of a scarce resource. `http://www.nvidia.com/object/gcr-energy-efficiency.html`. Accessed 12-April-2017.

[136] Michael Orlov and Moshe Sipper. Genetic programming in the wild: Evolving unrestricted bytecode. In *Proceedings of the 2009 Genetic and Evolutionary Computation Conference — GECCO '09*, pages 1043–1050. ACM, 2009.

[137] Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, 2011.

[138] Fernando Otero, Tom Castle, and Colin Johnson. Epochx: Genetic programming in Java with statistics and event monitoring. In *Proceedings of the 2012 Conference on Genetic and Evolutionary Computation Companion — GECCO Companion '12*, pages 93–100. ACM, 2012.

[139] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, 2016.

[140] Edoardo Paone, Francesco Robino, Gianluca Palermo, Vittorio Zaccaria, Ingo Sander, and Cristina Silvano. Customization of OpenCL applications for efficient task mapping under heterogeneous platform constraints. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition — DATE '15*, pages 736–741. IEEE, 2015.

[141] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. The influence of the Java collection framework on overall energy consumption. In *Proceedings of the 2016 International Workshop on Green and Sustainable Software — GREENS '16*. ACM, 2016.

[142] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. Genetic Improvement of software: A comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 2017.

[143] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. Specialising software for different downstream applications using genetic improvement and code transplantation. *IEEE Transactions on Software Engineering*, 2017.

[144] Justyna Petke, William B Langdon, and Mark Harman. Applying genetic improvement to MiniSAT. In *Proceedings of the 2013 International Symposium on Search Based Software Engineering — SSBSE '13*, pages 257–262. Springer, 2013.

[145] Justyna Petke, William B. Langdon, Mark Harman, and Westley Weimer. Using genetic improvement & code transplants to specialise a C++ program to a problem class. In *Proceedings of the 2014 European Conference on Genetic Programming — EuroGP '14*, pages 137–149. Springer, 2014.

[146] Sergio Pino, Lori Pollock, and Sunita Chandrasekaran. Exploring translation of OpenMP to OpenACC 2.5: lessons learned. In *Proceedings of the 2017 Parallel and Distributed Processing Symposium Workshops — IPDPSW '17*, pages 673–682. IEEE, 2017.

[147] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 2014 International Conference on Software Engineering — ICSE '14*, pages 254–265. ACM, 2014.

[148] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis — ISSTA '15*, pages 24–36. ACM, 2015.

[149] Conor Ryan, J. J. Collins, and Michael O. Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of the 1998 European Conference on Genetic Programming — EuroGP '98*, pages 83–96. Springer, 1998.

[150] Cagri Sahin, Lori Pollock, and James Clause. How do code refactorings affect energy usage? In *Proceedings of the 2014 Symposium on Empirical Software Engineering and Measurement — ESEM '14*, page 36. ACM, 2014.

[151] Harvey M Salkin and Cornelis A De Kluyver. The knapsack problem: a survey. *Naval Research Logistics Quarterly*, 22(1):127–144, 1975.

[152] Stanley A. Sawyer, John Parsch, Zhi Zhang, and Daniel L. Hartl. Prevalence of positive selection among nearly neutral amino acid replacements in drosophila. *Proceedings of the 2007 National Academy of Sciences of the United States of America — PNAS '07*, 104(16):6504–6510, 2007.

[153] Peter Schneider-Kamp, René Thiemann, Elena Annov, Michael Codish, and Jürgen Giesl. Proving termination using recursive path orders and SAT solving. In *Proceedings of the 2007 International Symposium on Frontiers of Combining Systems — FroCoS '07*, pages 267–282. Springer, 2007.

[154] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *Proceedings of the 2014 Conference on Architectural support for programming languages and operating systems — ASPLOS '14*, pages 639–652. ACM, 2014.

[155] Dominic P Searson, David E Leahy, and Mark J Willis. GPTIPS: an open source genetic programming toolbox for multigene symbolic regression. In *Proceedings of the 2010 International Multiconference of Engineers and Computer Scientists*, volume 1, pages 77–80. Citeseer, 2010.

[156] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 2011 Symposium on the Foundations of Software Engineering — FSE '11*, pages 124–134. ACM, 2011.

[157] Wellisson G. P. Silva, Lisane Brisolara, Ulisses B. Corrêa, and Luigi Carro. Evaluation of the impact of code refactoring on embedded software efficiency. In *Proceedings of the 1st Workshop de Sistemas Embarcados*, pages 145–150, 2010.

[158] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic Programming for shader simplification. *ACM Transactions on Graphics*, 30(6):152, 2011.

[159] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 Conference on Mining Software Repositories — MSR '05*, pages 24–28. IEEE, 2005.

[160] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Computing in science & engineering*, 12(3):66–73, 2010.

[161] Gilbert Syswerda. A study of reproduction in generational and steady state genetic algorithms. *Foundations of Genetic Algorithms*, 2:94–101, 1991.

[162] John Taylor. *Introduction to error analysis, the study of uncertainties in physical measurements*. University Science Books, 1997.

[163] The World Bank. Co_2 emissions. `http://data.worldbank.org/indicator/EN.ATM.CO2E.KT/countries`. Accessed: 10-January-2015.

[164] James Tiongson. Mobile app: Marketing insights, 2015. Accessed: 24-March-2017.

[165] Shigeyoshi Tsutsui, Masayuki Yamamura, and Takahide Higuchi. Multi-parent recombination with simplex crossover in real coded genetic algorithms. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference — GECCO '99*, pages 657–664. Morgan Kaufmann, 1999.

[166] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Opium: Optimal package install/uninstall manager. In *Proceedings of the 2007 International Conference on Software Engineering — ICSE '07*, pages 178–188. IEEE, 2007.

[167] Alan M. Turing. Intelligent machinery. 1948.

[168] Alan M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[169] Peter J. M. Van Laarhoven and Emile H. L. Aarts. Simulated annealing. In *Simulated Annealing: Theory and Applications*, pages 7–15. Springer, 1987.

[170] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the Conference on Computer Vision and Pattern Recognition — CVPR '01*. IEEE, 2001.

[171] Paul Viola and Michael Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, 2004.

[172] Paul Walsh and Conor Ryan. Automatic conversion of programs from serial to parallel using genetic programming — The Paragen system. In *Proceedings of the 1995 Parallel Computing Conference — ParCo '95*, pages 415–422. ACM, 1995.

[173] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 2013 International Conference on Automated Software Engineering — ASE '13*, pages 356–366. IEEE, 2013.

[174] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 2009 International Conference on Software Engineering — ICSE 2009*, pages 364–374. IEEE, 2009.

[175] August Weismann. Essays on heredity and kindred biological subjects, 1889.

[176] Peter A. Whigham. Grammatically-based genetic programming. In *Proceedings of the 1995 workshop on Genetic Programming: from theory to real-world applications*, volume 16, pages 33–41, 1995.

[177] Peter A Whigham, Grant Dick, James Maclaurin, and Caitlin A Owen. Examining the best of both worlds of grammatical evolution. In *Proceedings of the 2015 Genetic and Evolutionary Computation Conference — GECCO '15*, pages 1111–1118. ACM, 2015.

[178] David R. White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 2011.

[179] David R. White, John Clark, Jeremy Jacob, and Simon M. Poulding. Searching for resource-efficient programs: Low-power pseudorandom number generators. In *Proceedings of the 2008 Genetic and Evolutionary Computation Conference — GECCO '08*, pages 1775–1782. ACM, 2008.

[180] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC — First experiences with real-world applications. In *Proceedings of the 2012 European Conference on Parallel Processing — Euro-Par '12*, pages 859–870. Springer, 2012.

[181] Claas Wilke, Sebastian Richly, Sebastian Gotz, Christian Piechnick, and Uwe Aßmann. Energy consumption and efficiency in mobile applications: A user feedback study. In *Proceedings of the 2013 Conference on and IEEE Cyber, Physical and Social Computing — CPSCom '13*, pages 134–141. IEEE, 2013.

[182] Josh L. Wilkerson and Daniel Tauritz. Coevolutionary automated software correction. In *Proceedings of the 2010 Genetic and Evolutionary Computation Conference — GECCO '10*, pages 1391–1392. ACM, 2010.

[183] Sewall Wright. *The roles of mutation, inbreeding, crossbreeding, and selection in evolution.* 1932.

[184] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep parameter optimisation. In *Proceedings of the 2015 Conference on Genetic and Evolutionary Computation — GECCO '15*, pages 1375–1382. ACM, 2015.

[185] Kwaku Yeboah-Antwi and Benoit Baudry. Online genetic improvement on the java virtual machine with ECSELR. *Genetic Programming and Evolvable Machines*, 18(1):83–109, 2017.

[186] Mike Yi. Intel power gadget. `https://software.intel.com/en-us/articles/intel-power-gadget`, 2016. Accessed 12-June-2017.

[187] Shin Yoo. Embedding genetic improvement into programming languages. In *Proceedings of the 2017 Genetic and Evolutionary Computation Conference Companion — GECCO Companion '17*, pages 1551–1552. ACM, 2017.

[188] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A highly configurable cache architecture for embedded systems. In *Proceedings of the 2003 International Symposium on Computer Architecture — ISCA*, pages 136–146. IEEE, 2003.

[189] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. Search based requirements optimisation: Existing work and challenges. In *Requirements Engineering: Foundation for Software Quality*, pages 88–94. Springer, 2008.

Appendix A

OpenACC Grammar

```
<start>::= <start> <start> | <base>
<base>::= "#pragma acc " <choice>
<choice>::= "parallel "  <parallel> <private>
                <reduction> <optional_block> <line_number>
| "loop " <loop> <private> <reduction> <loop_line_number>
| "parallel loop " <loop> <parallel> <private> <reduction>
                <loop_line_number>
| "kernels " <kernels> <private> <block> <line_number>
| "kernels loop " <loop> <kernels> <private>
                <loop_line_number>
| "data " <data> <block> <line_number>
| "cache(" <variables> ") "  <top_loop_line_number>
| "atomic " <atomic_clause> <line_number>
| "update " <async> <wait> <update> <line_number>
| "routine " <routine> <function_line_number>
| "wait " <async> <line_number>
| "wait(" <sync_number> ") " <async> <line_number>
<parallel>::= <async> <wait> <num_gangs> <num_workers>
                 <vector_length> <data> <firstprivate>
<loop>:: = <gang> <worker> <vector> <seq> <collapse>
                <auto> <independent>
<kernels>::= <async> <wait> <data>
<data>::= <copy> <copyin> <copyout> <create> <present>
                <present_or_copy> <present_or_copyin>
                <present_or_copyout> <present_or_create>
<routine>::= <gang_singular><worker_singular> <vector_singular>
                <seq>
<update>::= "self(" <variables> ") "
| "host(" <variables> ") "
| "device(" <variables> ") "
| " "
```

```
<async>::= "async "
| "async(" <sync_number> ")"
| " "
<wait>::= "wait "
| "wait(" <sync_number> ")"
| " "
<sync_number>::= "1" | "2" | "3" | "4" | "5"
<num_gangs>::= "num_gangs(" <two_power> ") "
| " "
<num_workers>::= "num_workers(" <two_power> ") "
| " "
<vector_length>::= "vector_length(" <two_power> ") "
| " "
<reduction>::= "reduction(" <reduction_operator> ":" <variables> ")"
| " "
<reduction_operator>::= "+" | "*" | "max" | "min" | "&"
                        | "|" | "^" | "&&" | "||"
<copy>::= "copy(" <variables> ") "
| " "
<copyin>::= "copyin(" <variables>") "
| " "
<copyout>::= "copyout(" <variables> ") "
| " "
<create>::= "create(" <variables> ") "
| " "
<present>::= "present(" <variables> ") "
| " "
<present_or_copy>::= "present_or_copy(" <variables> ") "
| " "
<present_or_copyin>::= "present_or_copyin(" <variables> ") "
| " "
<present_or_copyout>::= "present_or_copyout(" <variables> ") "
| " "
<present_or_create>::= "present_or_create(" <variables> ") "
| " "
<private>::= "private(" <variables> ") "
| " "
<firstprivate>::= "firstprivate(" <variables> ") "
| " "
<collapse>::= "collapse(" <collapse_number> ") "
| " "
<collapse_number>::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
                     | "9" | "10"
<gang>::= <gang_singular>
| "gang(" <two_power> ") "
<gang_singular>::= "gang " | " "
```

```
<worker>::= <worker_singular>
| "worker(" <two_power> ") "
<worker_singular>::= "worker " | " "
<vector>::= <vector_singular>
| "vector(" <two_power> ") "
<vector_singular>::= "vector " | " "
<seq>::= "seq "
| " "
<auto>::= "auto "
| " "
<independent>::= "independent "
| " "
<atomic_clause>::= "read "
| "write "
| "capture "
| "update " <optional_block>
<optional_block>::= <block>
| " "
<two_power>::= "2" | "4" | "8" | "16" | "32" | "64" | "128" | "256"
                 | "512" | "1024"
<block>::= "\n{ " <block_placeholder>
<variables>::= <variable>
| <variable> "," <variables>
<variable>::= <variable_placeholder>
<block_placeholder>::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
| "9" | "10" | "11" | "12" | "13" | "14" | "15" | "16" | "17" | "18"
| "19" | "20" | "21" | "22" | "23" | "24" | "25" | "26" | "27" | "28"
| "29" | "30" | "31" | "32" | "33" | "34" | "35" | "36" | "37" | "38"
| "39" | "40" | "41" | "42" | "43" | "44" | "45" | "46" | "47" | "48"
| "49" | "50" | "51" | "52" | "53" | "54" | "55" | "56" | "57" | "58"
| "59" | "60" | "61" | "62" | "63" | "64" | "65" | "66" | "67" | "68"
| "69" | "70" | "71" | "72" | "73" | "74" | "75" | "76" | "77" | "78"
| "79" | "80" | "81" | "82" | "83" | "84" | "85" | "86" | "87" | "88"
| "89" | "90" | "91" | "92" | "93" | "94" | "95" | "96" | "97" | "98"
| "99" | "100"
<variable_placeholder>::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
| "9" | "10" | "11" | "12" | "13" | "14" | "15" | "16" | "17" | "18"
| "19" | "20" | "21" | "22" | "23" | "24" | "25" | "26" | "27" | "28"
| "29" | "30" | "31" | "32" | "33" | "34" | "35" | "36" | "37" | "38"
| "39" | "40" | "41" | "42" | "43" | "44" | "45" | "46" | "47" | "48"
| "49" | "50" | "51" | "52" | "53" | "54" | "55" | "56" | "57" | "58"
| "59" | "60" | "61" | "62" | "63" | "64" | "65" | "66" | "67" | "68"
| "69" | "70" | "71" | "72" | "73" | "74" | "75" | "76" | "77" | "78"
| "79" | "80" | "81" | "82" | "83" | "84" | "85" | "86" | "87" | "88"
| "89" | "90" | "91" | "92" | "93" | "94" | "95" | "96" | "97" | "98"
| "99" | "100"
```

Appendix B

Deep parameter tuning buggy code

```
if(numEvaluations < populationSize)
{
        for(int i=0; i<solution.getNumberOfVariables(); i++)
        {
                int val = genotype[i];
                if(i == ((numEvaluations
                                % solution.getNumberOfVariables()) - 1))
                        val += ((numEvaluations - 1)
                {
                                / solution.getNumberOfVariables() + 1);
                solution.getVariable(i).setValue(val);
                }
        }
}
```

Figure B.1: Fixed Implementation

```
for(int i=0; i < solution.getNumberOfVariables(); i++)
{
        int val=genotype[i];
        if(i == ((numEvaluations
                        % solution.getNumberOfVariables()) - 1))
        {
                val +=((numEvaluations -1)
                        /solution.getNumberOfVariables() + 1);
        }
        solution.getVariable(i).setValue(val);
}
```

Figure B.2: Faulty Implementation

Appendix C

DawnCC Investigation

DawnCC is a compiler module which automatically detects parallelisable code in C/C++ programs and then inserts OpenACC or OpenMP directives where appropriate [125]. We carried out a small investigation to determine DawnCC's performance when inserting OpenACC directives on the Seoul National University NAS Parallel Benchmark suite (SNU-NPB) [50]. The SNU-NPB suite contains sequential versions of seven applications known to be parallelisable using OpenACC [146].

C.1 Experiment Setup

Our experiment to evaluate DawnCC's performance was quite simple. For each application within the SNU-NPB suite we ran DawnCC [1] to produce a variant which contained OpenACC directives. We ran both the sequential and DawnCC variant of each application 100 times, on a test case (selection discussed below). We then compared the means of these runs, and whether they were statistically significant (using the Wilocoxon Rank Sum test) to determine DawnCC's effectiveness.

Table C.1 shows the apps from the SNU-NPB suite and the test cases we used to evaluate DawnCC against for each application. The applications in the SNU-NPB suite each have a set of input classes (input data with corresponding output data; essentially a black box test). To evaluate

[1] As available from the DawnCC GitHub [3] on the 7th of November 2017.

Application	Input class
BT	Custom*
CG	Custom*
EP	A
FT	A
LU	W
MG	B
SP	W

Table C.1: The SNU-NPB applications targeted and the test case used for evaluation. *Custom test case defined in text.

Application	Sequential Mean (s)	DawnCC Mean (s)	Wilcoxon Rank Sum Test p-value
BT	15.35	73.20	$\ll 0.001$
CG	8.43	11.26	$\ll 0.001$
EP	21.92	27.65	$\ll 0.001$
FT	8.54	8.51	0.670
LU	5.77	8.76	$\ll 0.001$
MG	8.06	8.17	0.005
SP	5.69	5.74	0.009

Table C.2: DawnCC's performance on the SNU-NPB suite's sequential implementation.

the original and the DawnCC variant, we selected the input class that ran in greater that 5 seconds (so that smaller reductions in execution time could be detectable and not confused with statistical variance) and less than 30 seconds (in the interests of keeping experiment times low). If two or more input classes fell within this range then the one with the lowest execution time was chosen. If there were no input classes a custom input class was created.

Both BT and CG required custom classes to be created. A problem class for BT was created with 40x40x40 grids over 200 time steps with DT equal to 0.8×10^{-3}.For CG we setup a problem class with a size of 30,000 over 30 iterations.

Experiments were carried out on an Ubuntu 14.04.5 LTS Desktop system with an Intel Core i5-650 processor (3.2 GHz, 2 cores), 4GB of RAM and an nVidia GeForce GTX 1060 GPU. The code was compiled using the PGI 17.4-0 C compiler.

C.2 Experiment Results

Table C.2 shows the results from the experiments. We found that in no case did the DawnCC variant decrease execution time. Of the seven applications studied, six are found to increase execution by a statistically significant extent ($p < 0.01$), with the the SP DawnCC variant having no statistically significant influence on execution time.